FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Large-scale Program Analysis for Language Evolution

by

Filip Křikava

A habilitation thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague

Bucaramanga, December 2020

Programming Language Laboratory
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Head of department: prof. Jan Vitek

Abstract

Programming languages need to evolve constantly otherwise they fall out of favor, become neglected and lost. The hard part of growing a language is to make the changes as little disruptive as possible. Each change has to be carefully reviewed for its impact on the ecosystem. However, until recently, language designers and engineers had only a few means to understand the impact of a programming language change. The cloud code-hosting websites have changed that by making code a shared resource and giving everyone access to a huge number of open-source projects. That has opened whole new opportunities in language evolution. We can use program analysis to get empirical evidence about how language is used in real-world code.

This thesis is based on the published research done at the PRL-PRG research laboratory since summer 2017. It presents our experience in *conducting large-scale program analyses of public code repositories*. Concretely, we include three different analyses with which we try to answer the following questions: (1) *How well can automated trace-based unit test extraction actually work in practice for R?* The aim is to reduce the burden of writing a comprehensive unit test suites in the cases where it may be possible for a tool to extract them automatically from a client code. (2) *What expressive power do we need to ascribe types to R function?* The goal is to retrofit a type system in R that would benefit the users in making the code more reliable and increase our assurance data analysis for which R is used so much. (3) *How are Scala implicits used in the wild?* The aim is to provide empirical evidence on the use and misuse of this distinct Scala feature.

To answer this, we have developed a complete toolchain for doing dynamic program analysis in R and a static analyzer that can extract implicit usage from Scala code. We then evaluated the program analyses on large corpora spanning millions of lines of R and Scala code. The results are: (1) A tool that increased the test code coverage from 19% to 53% on a corpus of 1,500 R packages. (2) A type language, which found a compromise between simplicity and usefulness, that was used to ascribe type signatures to 25,000 functions from the most widely used R packages. (3) A study that is both a retrospective on introducing implicits into the wild and a means to inform designers of future languages of how people use and misuse them.

Keywords: program analysis, language evolution, test extraction, program tracing, type declarations, dynamic languages, corpora analysis, implicit parameters, implicit conversion, R, Scala.

Acknowledgements

I want to thank my collaborators, the members of the PRL-PRG lab, and especially prof. Jan Vitek. I wish everyone could have a mentor like Jan. Moving from self-adaptive software systems into programming language research has been challenging, and he has provided just the perfect environment and guidance!

Finally, I want to thank my family for their encouragement and support, Johana for her love, and Alicia for her smiles and all the happiness and joy she brings!

Contents

CHAPTER 1

Introduction

1.1 Context and Motivation

Much like natural languages that allow us to talk to one another, programming languages give us a way to communicate with computers. Since the inception of the lambda calculus in the 1930s, thousands of languages have been created.[1] The majority of these are no longer relevant, had been deprecated, superseded, or simply fell out of favor, but many are still in daily use. And much like natural languages, they also need to evolve constantly. If they do not, they will follow the fate of Ada, Algol, Pascal, or Smalltalk, languages that once were popular but eventually became neglected and lost.

Programming language evolution. There are different forces behind programming language evolution [Urma, 2017]. On the one hand, they need to track the changes in the hardware evolution and the trends in both the industry and academia. For example, in recent years, we have seen a continuous adoption of functional-programming concepts into traditionally object-oriented imperative programming languages such as C++ (*e.g.*, $\lambda-$functions in C++11, polymorphic $\lambda-$functions in C++14, generic programming using concepts in C++20) or Java (*e.g.*, parametric-polymorphism in Java 5, $\lambda-$functions with map-reduce inspired stream API in Java 8, pattern-matching and records in Java 14). On the other hand, languages also need to deal with design shortcomings and bugs. That includes mistakes in the API or simplifying commonly recurring verbose code patterns. For example, in version 5, Java added a *for-each* construct, which significantly reduced the necessary setup for collection traversals. In version 6, the diamond operator improved the type inference for generic instance creation, and in version 10, the new `var` keyword introduced local-type inference.

The hard part of growing a programming language is to make the changes as little disruptive as possible. As Martin Buchholz, who has been maintaining JDK over the last two decades, puts it, *"Every change is an incompatible change. A risk/benefit analysis is*

[1] Diarmuid Pigott's Online Historical Encyclopaedia of Programming Languages (`hopl.info`) lists over 8,000 programming languages.

always required" [Darcy, 2008]. Not only the compiler and possibly the runtime system need to be changed, but users need to be notified, libraries updated and documentation invalidated including books and *stackoverflow.com*-like discussion forum. For instance, before Java moved to the 6-months released cycle in 2018, Oracle, the main Java sponsor, estimated the cost of every new release to $50M. There are over 5K books, 1.7M questions on *stackoverflow.com*, and 3.5M public repositories on *github.com*, which all become less relevant with every new Java version unless updated.

Sometimes, the change (or the changes) becomes too big or too disturbing that it spawns a new major version of the language that is partially backward-incompatible with the old one. This is the most drastic way of a language evolution and puts a substantial load to the whole community. The canonical example is the Python 2 to Python 3 transition, which has been ongoing for the past 12 years. Even though Python 2 is no longer maintained it is still actively used[2] and thus many of the libraries and applications still need to support both versions. This comes with a significant maintenance overhead (*e.g.*, releasing libraries for the two versions, back-porting Python 3 features into Python 2).

Another example is Scala. With the experience from Scala 2, the team at EPFL has redesigned the language, and the resulting Scala 3 shall be released next year. While the new language tries to be mostly compatible with the old one, some migration effort will be needed. Therefore, part of the community is wondering if the transition to Scala 3 will be similarly slow and painful as the Python's was. It is hard to predict the adoption of the new language, but two things are different this time. First, Scala is a statically typed language. The type checker can catch many errors at compile time. Moreover, thanks to static typing, large portions of code can be automatically rewritten using static analysis tools such as SCALAMETA[3] or SCALAFIX[4]. Second, and perhaps more important, we now have access to an incredible amount of code.

Big code. Until recently, language designers and implementers had few means to understand the impact of a change in programming language. They had to rely on community surveys, mailing lists, and intuition to assess a language change. The cloud code-hosting websites such as GitHub, GitLab, or BitBucket have changed that. They made code a shared resource giving everyone access to a huge number of open-source projects written in all possible languages. For example, GitHub itself has over 30 million users and hosts more than 100M projects.[5] This opens whole new opportunities in language evolution because we can get empirical evidence from real-world code. We can ask how much code will be affected by deprecating some language feature or library function. We can look for patterns and how certain language features are used to figure out if there is perhaps a better way to achieve the same. In general, we can mine these large code repositories for insights about how language features or library API is

[2] https://www.jetbrains.com/lp/python-developers-survey-2019/
[3] Library to read, analyze, transform and generate Scala programs, *cf.* https://scalameta.org/
[4] Rewrite and linting tool for Scala, *cf.* https://github.com/scalacenter/scalafix
[5] https://github.blog/2018-11-08-100m-repos/

used in the real world. This can help us better estimate the risks/benefits that Martin Buchholz talked about.

For example, in the case of Scala, we no longer need to guess how many projects cannot be migrated automatically. Instead, we can get hundreds of thousands of Scala projects just from GitHub and try.

Analyzing code repositories. Having access to source code repositories with real-world projects is a gold mine for language researchers and engineers. Using program analysis to mine these repositories is an increasingly important research topic. In general, there are two complementary approaches [Ernst, 2003]. Static program analysis, which reasons about programs behavior using information from the programs' code without actually running them [Møller and Schwartzbach, 2018] and dynamic program analysis that relies on execution traces to get insights about code [Ball, 1999]. The essential difference is that static analysis tries to approximate all possible program executions while dynamic analysis reasons about program behavior under a specific workload and input. For static analysis, we usually need just the source code; for dynamic analysis, we need runnable code with input.

Program analysis has been used to study a wide range of topics in public code repositories. In the context of the Java programming language, for example, researchers were looking at how method chaining is used and how to improve it [Nakamaru et al., 2020], how program scale affects Java projects [Lopes and Ossher, 2015], how reflection is used and how to help static analysis to resolve it [Landman et al., 2017; Barros et al., 2015], how language features are adopted over time [Dyer et al., 2014], how unsafe API is used and how to make it safer [Mastrangelo et al., 2015; Huang et al., 2019], or using dynamic analysis to check the correct usage of Java APIs [Legunsen et al., 2016], and identify Java task-parallel workloads suitable for inclusion in a domain-specific benchmark suite [Villazón et al., 2019]. With the massive spread of web applications, JavaScript has also received a lot of attention, despite being notably hard to analyze due to its dynamic behavior [Richards et al., 2010] and tight coupling with DOM and Browser API [Jensen et al., 2011]. For example, program analysis was used to study how `eval` is used [Richards et al., 2011b] and how to remedy it [Meawad et al., 2012; Jensen et al., 2012], how to predict identifier names and type annotations [Raychev et al., 2015], how to detect race conditions in web applications [Adamsen et al., 2018], how new language constructs such as promises are adopted [Villazón et al., 2019], or how to detect and test breaking library changes [Mezzetti et al., 2018; Møller and Torp, 2019; Nielsen et al., 2020].

Some languages received less attention. A notable example is the R programming language that has been largely neglected by the programming language and software engineering communities. That leaves much unrealized potentials, which makes it an attractive research target. On the other hand, program analysis of R is not easy. First, similarly to JavaScript, R also exhibits extensive dynamic behavior, which makes it a difficult target for program analysis [Morandat et al., 2012]. Since the language researchers

have mostly overlooked it, there is practically no tooling or infrastructure for doing program analysis of R code. Building such support invariably entails substantial engineering effort. Finally, much of our intuition trained on working with general purpose programming languages may fail us when dealing with domain-specific languages such as the R data science language. In this case, it is only more crucial to look at how users work with the language to make design decisions that benefit them.

This thesis. In this thesis, we present our experience in *conducting large-scale program analyses of public code repositories*. We present three different analyses: two in the context of the R programming language and one in Scala. Our group's main research target is R, and the work on Scala came mainly from the thesis's author personal interest. The two languages are quite different from one another, and so were the studies, which allowed us to gain experience in different contexts. In the case of R, we used a dynamic analysis, while for Scala we did a static analysis. R has centralized, curated repository of libraries while for Scala we used the code from GitHub. For R we had to develop the entire infrastructure from scratch. For Scala, we could rely on the compiler and its existing plugin infrastructure and tools. On the other hand, they share a general notion of a large scale analysis. For both we had collected a corpus of projects and assembled a flexible and scalable data analysis pipelines that can handle millions lines of code.

1.2 Summary of Contributions

This thesis provides an overview of the research that we have done between summer 2017 — summer 2020 at the programming language research laboratory[6] that is spread between Czech Technical University in Prague and Northeastern University in Boston and is led by prof. Jan Vitek. Concretely, we will present three large-scale program analysis studies that aim to answer questions related to the use of programming languages in the wild. Two in the context of the R programming language which has been in the spotlight of this research group. The last one for the Scala programming language in collaboration with Heather Miller from Carnegie Mellon University.

In this section, we provide a summary of the key contributions done in each paper.

1.2.1 Tests from Traces: Automated Unit Test Extraction for R

By *Filip Křikava and Jan Vitek*, published in *Proceedings of the 27th ACM International Symposium on Software Testing and Analysis (ISSTA)*, August 2018 [Křikava and Vitek, 2018a].

In this paper we looked into how well unit tests for a target software package can be extracted from execution traces of client code, concretely for the R programming language. The objective is to reduce the effort involved in creating test suites while minimizing the

[6] *cf.* https://prl-prg.github.com

number and size of individual tests, and maximizing coverage. The paper made the following contributions:

- We implemented a tool, GENTHAT [7] for automated extraction of unit tests for R.
- We designed a data analysis pipeline for an empirical evaluation of that tool on a large corpus of 1,545 packages with 1,709,796 lines of code.
- We demonstrate that it is possible to improve code coverage significantly: (1) On average, the default tests that come with the packages cover only 19%. (2) After deploying GENTHAT we can increase the coverage to 53%. This increase mostly comes from extracting tests from all the available executable artifacts in the package and the artifacts from packages that depend on this package. (3) GENTHAT is surprisingly accurate. It can reproduce 80% of the calls executed by the scripts. (4) It can also greatly reduce the number and size of test cases retained in the extracted suite, running 1.9 times faster than package examples, tests and vignettes combined.

The paper was submitted with an artifact [Křikava and Vitek, 2018b] which got the ACM evaluated artifact—*reusable* badge and received the *distinguished artifact award*. The tool, GENTHAT, was also presented to the R practitioners at the international *UseR!*[8] conference in 2018 [Krikava, 2018].

1.2.2 Designing Types for R, Empirically

By *Alexi Turcotte, Aviral Goel, Filip Křikava and Jan Vitek* published in *Proceedings of the ACM Programming Languages 4, OOPSLA*, Article 181, November 2020 [Turcotte et al., 2020c].

In this paper, we looked into how to retrofit a type system into R, a programming language oriented towards interactive and exploratory programming style with poor specification and eclectic mix of features (*cf.* Section 2.1). This is a part of our outgoing effort to eventually propose a type system for inclusion in the language. For this work, we limited the scope of our investigation to ascribing types to function signatures. The paper made the following contributions:

- We designed a simple type language that found a compromise between simplicity and usefulness by focusing on R's most widely used features.
- We implemented scalable and robust tooling to automatically extract type signatures (TYPETRACER) and instrument R functions with checks based on their declared types (CONTRACTR).
- We carried out a program analysis of 412 widely used and maintained packages to synthesize type signatures for 25,215 functions and validated the robustness of these signatures against 160,379 programs that use those functions.
- We report on the appropriateness and usefulness of a simple type language for R. Overall, we found our design to fit quite well with the existing language: (1) Nearly

[7] *cf.* https://github.com/PRL-PRG/genthat
[8] The largest venue for users of the R language.

80% of functions are either monomorphic or have only one single polymorphic argument. (2) When we tested the types inferred by TYPETRACER during our evaluation, we found that less than 2% of contract assertions failed. (3) Furthermore, we found that our type language and contract checking framework would help programmers eliminate or otherwise simplify 61% of existing type checks and assertions in user code. In summary, we believe that our simple type language design is a solid foundation for the eventual type system for R.

The paper was submitted with an artifact [Turcotte et al., 2020a] which got the ACM evaluated artifact—*reusable* badge and the complete data set was made publicly available [Turcotte et al., 2020b].

1.2.3 Scala Implicits are Everywhere: A Large-scale Study of the Use of Scala Implicits in the Wild

By *Filip Křikava, Heather Miller and Jan Vitek* published in *Proceedings of the ACM Programming Languages 3, OOPSLA*, Article 163, October 2019 [Křikava et al., 2019].

In this paper, we report on a large-scale study of the use of the Scala implicit parameters and implicit conversions in the wild. They represent a distinctive language feature that provides a way to reduce the boilerplate code and implement certain language features outside of the compiler. While powerful, they have some flaws (misuse, complexity, and compilation time overhead). This study is both a retrospective on the result of introducing this feature into the wild and a means to inform designers of future language of how people use and misuse implicits. It makes the following contribution:

- We implemented scalable data analysis pipeline for running static data analysis tools over Scala semantic data.
- We have assembled a large corpus of 7,280 Scala open-source projects hosted on GitHub with 18,713,247 lines of code. At the time of writing, it has been the largest corpus of compiled Scala projects to the best of our knowledge.
- We built a static analyzer that extracts implicit declarations and calls sites from semantic data into a model that can be queried.
- We provide the results of the analysis: 98.2% of projects have at least one implicit call (1) *Scala implicits are everywhere* as site, and 5,694 contain at least one implicit definition. Across the projects, median of the implicit call sites' ratio is 23.4%—*i.e.*, *one out of every four calls sites involves implicits*. (2) Most projects do not misuse the implicits. We quantify the use of six common patterns and two anti-patterns. (3) Most implicit call sites take between 0-2 implicit parameters. However, this gets much higher in the libraries that rely on implicit type-class derivation. (4) Finally, we observe slower compilation times for projects that use implicit type-class derivation.

The paper was submitted with an artifact [Krikava et al., 2019] which got the ACM evaluated artifact—*reusable* badge.

1.3 Thesis Outline

We organize the rest of this thesis as follows: Chapter 2 starts with a background on the R programming language, presents the tools that we have built for analyzing R code, discusses our approach to building program analysis pipelines, and finally, overview work on program analysis related to this thesis. We focus primarily on R because in Scala's case, we could build on the existing tools, and unlike R, Scala comes from a programming language research laboratory and thus is more known to our community. Next, in Chapter 3, we present the contribution of the three papers included in this thesis about automated unit test extraction for R packages in Section 3.1, empirically designing a type language for R in Section 3.2, and studying the use of implicit in the Scala programming language in Section 3.3. We conclude this thesis and outline further research in Chapter 4. We enclose the individual published papers in Appendix A.

CHAPTER 2

Background

In this chapter we start with a short primer on the relevant characteristics of the R programming language needed for the two R program analysis (*cf.* Sections 3.1 and 3.2). Next, we describe the tooling and the necessary infrastructure that we have developed for analyzing R, point out why analyzing R is hard and we discuss our approach to building program analysis pipelines. Finally, we discuss related work.

2.1 The R Programming Language

R is the language for statistics and data science. It was designed in 1993 by statisticians Ross Ihaka and Robert Gentleman [Ihaka and Gentleman, 1996] as an alternative implementation to the S programming language [Becker et al., 1988] created by John Chambers in the 1970' at Bell Labs. At the language level, the main change was the introduction of lexical scoping semantics inspired by Scheme and a garbage collector. At the community level, the breaking change was the release of the language under the open-source GNU public license in 1995. Over the years, R was designed, implemented, and maintained by statisticians. In the beginning, it was developed as a glue languages for statistical routines written in Fortran. Today, R is one of the key tools for sophisticated data analysis in wide range of fields. It is oriented towards data scientists—*i.e.*, non-programmers with a focus on rapid prototyping and interactive use. The R-consortium estimates about 2M of R users worldwide[1]. Industrial use is large in companies ranging from Google, which has developed its own style guide[2], to Microsoft, who provides its own R distribution based on GNU R[3].

2.1.1 The language

R is a surprisingly rich language with a rather unusual mixture of features: it is vectorized, dynamically typed, lazy functional language with limited side-effects, extensive

[1] *cf.* https://www.r-consortium.org/about
[2] *cf.* https://google.github.io/styleguide/Rguide.html
[3] *cf.* https://mran.microsoft.com/open

reflective facilities with first-class environments, and retrofitted multiple object-oriented programming support [Morandat et al., 2012]. That makes it a challenging target for program analysis:

- R does not have type annotations or a static type system. This means there is nothing to suggest what the expected arguments or return values of a function could be, and thus there is little to guide test generation.
- Symbols such as `+` and `(` can be redefined during execution. This means that every operation performed in a program depends on the state of the system, and no function call has a fixed semantics. This holds even for control flow operations such as loops and conditionals. While redefinitions are not frequent, they do occur, and tools must handle them.
- R does not distinguish between scalars and vectors (they are all vectors). A vector can hold either logical, integer, double, complex, character values or raw bytes. There is no intrinsic notion of subtyping in R, but in many contexts built-in types are automatically and silently coerced from more specific types to more general types when deemed appropriate. Some odd conversions occur in corner cases, such as `1<"2"` holds and `c(1,2)[1.6]` returns the first element of the vector. All vectorized data types are monomorphic, except `list` which can hold values of any type. For all monomorphic data types, attempting to store a differently-typed value will cause a conversion: either the value is converted to the type of the vector, or vice versa.
- R is fully reflective. It is possible to inspect any part of a computation (*e.g.*, the code, the stack, or the heap) programmatically. Moreover almost all aspects of the state of a computation can be modified (*e.g.*, variables can be deleted from an environment and injected in another).
- All expressions are evaluated by-need. For example, the call `f(a+b)` contains three delayed sub-expressions, one for each variable and one for the call to plus. This means that R does not pass values to functions but rather passes unevaluated promises (the order of evaluation of promises is part of the semantics as they can have side effects). These promises can also be turned back into code by reflection.
- Most values are vectors or lists. Values can be annotated by key-value pairs. These annotations, coupled with reflection, are the basic building blocks for many advanced features of R. An example of this are the four different object systems that use annotations to express classes and other attributes.
- R has a copy-on-write semantics for shared values. A value is shared if it is accessible from more than one variable. This means that side effects that change shared values are rare. This gives a functional flavor to large parts of R.
- There are multiple object-oriented systems: S3, S4, R5 (also called RC), and R6. The S3 object system supports single dispatch on the class of the first argument of a function, whereas the S4 object system allows multiple dispatch. R5 allows users to define objects in a more imperative style. R6 is a rewrite of R5 with more OOP features such as support for private and public fields and methods. Additionally, some packages,

for example, `ggplot2`[4] implements their own object-oriented system.

2.1.2 R Code Repositories

There are multiple large open-source code repositories for R. (1) The Comprehensive R Archive Network (*CRAN*)[5], which is the primary and largest repository of R code, hosts almost 16K packages and keeps growing with about six new package submissions a day [Ligges, 2017]. (2) The Bioconductor project[6], which provides tools for the analysis and comprehension of high-throughput genomic data, contains close to 2K R packages. (3) The *Kaggle* data science competition website[7] contains over 7.9K unique solutions in R to over 200 data analytic competitions. (4) Finally, GitHub reports more than 231K R projects.

Unlike sites like Kaggle and GitHub, where anyone can submit any code, both Bioconductor and CRAN are curated repositories. Each program deposited in the archive must come with documentation and abide by a number of well-formedness rules that are automatically checked asserting certain quality[8]. Most relevant for this work is that all of the runnable code is tested and only a successfully running package is admitted in the archive. There are three sources of runnable code that come with each CRAN package: *tests*, examples and *vignettes*. They are, respectively, traditional unit tests, code snippets from the documentation, and long-form use-cases written in Latex or Rmarkdown with executable snippets of R code.

In the studies presented in this thesis, we work with both CRAN and Kaggle. The intent here is to contrast code written by experienced R developers (CRAN) with code authored by typical end-users of the language (Kaggle). There are 4.6M lines of runnable code that can be extracted from the CRAN packages' examples, tests and vignettes, and 665.8K lines of runnable code in Kaggle programs.

2.1.3 Analyzing R code

The highly dynamic nature of R enable programming idioms that make it nearly impossible to derive reliable insights statically. On the other hand, given that there exists a large number of runnable R code makes the language well-suited for dynamic analysis. However, there was no support for dynamic tracing functionality in R other than the coarse-grained built-in `trace` function, which traces only the R closures' entry and exit points. To this end, we have built two reusable tools: R-DYNTRACE for the low-level tracing of R programs and RUNR, an R package containing a collection of utilities for building data-intensive dynamic analyses.

[4] Popular plotting library *cf.* https://ggplot2.tidyverse.org/
[5] *cf.* https://cran.r-project.org/web/packages/index.html
[6] https://www.bioconductor.org/packages
[7] *cf.* https://kaggle.com
[8] Moreover, all the executable artifacts from all CRAN packages are checked for each new release of the language. The core R developers actively reach out to package maintainers when they find compatibility issues.

R-dyntrace. The R-DYNTRACE project initially started as an extension to GNU R virtual machine with a set of `dtrace`-compatible[9] probes triggering on specific program execution events (*e.g.*, function entry and exit, promise creation and execution, eval entry and exit). Eventually[10], we dropped the `dtrace` compatibility and introduced a C API so an analysis could be written as an R package. Later, Aviral Goel made it compatible with R 3.5 and 4.0, and significantly extended its capabilities adding support for garbage collection events, S3/S4 dispatch, variable definition and mutation, and long-jumps used by the interpreter to implement non-local exit. The tool was presented at the *R Implementation, Optimization and Tooling* workshop (associated with UseR! conference) in 2019 [Goel et al., 2019]. So far, it has been used for two large program analyses: the study of lazyness [Goel and Vitek, 2019] and for R type system design [Turcotte et al., 2020c] presented in Section 3.2.

Actual dynamic analysis is written as a standalone R package. The package contains code that defines tracer state and registers callbacks for the relevant events. R-DYNTRACE API exports utilities to access the interpreter state, such as the evaluation order of built-in function arguments and variable lookup without triggering any registered callbacks. Furthermore, R-DYNTRACE can detect nesting in callbacks, resulting from the tracing algorithm inadvertently executing R code that potentially modifies the program state. This is a common stumbling block in naïve tracing attempts, and our design captures these cases.

The main drawback of R-DYNTRACE is that despite being a small extension to the GNU R virtual machine (currently it is 2K lines of C code), it is still an extension that needs to be updated with every new R version. R is not a small project. The latest 4.0 release spans over 300K lines of C, 100K lines of Fortran, and 270K lines of R code. That makes it tricky to ensure the probes are in all the right places. There is an outgoing discussion with the core R team about integrating R-DYNTRACE into GNU R under a compilation flag. However, precisely because of the maintenance issue, there has been a little move forward towards this goal. This is also one of the reason why the GENTHAT tool (*cf.* Section 1.2.1) was not implemented on the top of R-DYNTRACE. We want the tool to be used by ordinary R package developers without any extra dependency on a modified R virtual machine. Also, in this case, it was possible to implement the tracing without the need of a low-level interpreter state.

Runr. The package facilitates constructing multistage pipeline for analysis of tracing data using the map-reduce programming model. It contains utilities for solving the recurring tasks such as metadata and runnable code extraction, computing code coverage including reverse-code coverage, and running R code in isolation, simulating the CRAN checking environment as close as possible.

[9] A performance analysis and troubleshooting tool for Solaris, OSX and FreeBSD `http://dtrace.org/blogs/about/`. The reason why we have stated with dtrace was the ability to trace all the way to system calls in one framework.

[10] When we realized that there is no reliable `dtrace` implementation for Linux.

Challenges. Even with the above tools, dynamic tracing of R code is hard. The details and their proverbial devil are surprisingly tricky to get right at scale:

- R is a lazy language. Even a simple lookup forcing a promise might fire a large cascade of events. This makes it hard to relate low-level traced data to high-level insights.
- R implementation is intertwined between C and R making it difficult to properly separate events triggered by user code from the ones coming from language internals.
- R has evolved organically over the last three decades. Without clear specs, it has not always been clear the separation between public and private interface making it difficult to deprecate APIs. A good example is the functions that manipulate R environments. At the C level, there are over 300 such functions. At the R level, the `envir` parameter which identifies an environment to be manipulated can be a list, a positive integer, a negative integer, a string, an actual environment, or any other object for which there exists an implementation `as.environment` S3 method. This poses a significant cognitive overhead.
- Often, we want to summarize events happening on R objects for which we use model objects. The problem is that R simply allocates too many objects. This is because we are running real programs that often deal with large data. Therefore, we have to setup proper hooks in the garbage collector to deallocate appropriately and make sure we do not count anything twice.
- Cross-language boundaries can also be tricky. R packages contain a significant portion of native code, which can call back to R. Without care it might result in recursive hook invocation.
- Running a large amount of real-world code means that we often run into unexpected corner cases. These are mostly engineering issues but can quickly leave us deep in a rabbit hole.

2.2 Program Analysis Pipelines

In this section we describe our approach to building data analysis pipelines.

Corpus selection. Analyzing large codebases brings challenges. Seemingly simple things such as code selection suddenly become much more demanding. How do we select the representative code and make sure there is no duplication and bias? For example that most programs do not come from student projects. In the case of a hundred projects, it is easy, but it is much harder in the case of hundred thousand projects.

Regarding GitHub, Kalliamvakou et al. [2014] suggests a simple rule of thumb: at least three contributors. While this helps to filter out some personal or irrelevant projects, in our Scala implicit study (*cf.* Section 3.3) we found that the corpus contains a considerable code duplication. The problem is that even without forks, the corpus contained "unofficial" forks—*i.e.*, copies of source code with no metadata suggesting their

origin. To illustrate the severity, the corpus contained 102 copies of Apache Spark[11]. Since Spark is the largest Scala project (over 100K lines of code), keeping them would significantly skew the subsequent analysis as 37.6% of the entire data set would be identical. We used DÉJÀVU [Lopes et al., 2017] to help us identify file-level duplicates. But still, we spent a considerable effort to clean the corpus. In the end, we have excluded over half of the downloaded source code but lost fewer than 2.8% of GitHub stars. While the number of GitHub stars does not necessarily reflect project's quality, originals tend to have higher star counts than copies. Similarly, active projects usually have more stargazers than student projects. Deduplication is equally important when working with Kaggle where we found that 2.3K out of 10K were whole-file duplicates.

Pipeline. From our experience, extracting insights from code is an iterative process. We update the analysis as we learn from the data. The problem is that running it on a large code data set can take days and produce a vast amount of data. Therefore, the pipeline has to be robust to handle the load yet flexible to allow for the exploratory nature of these studies.

We use a simple approach to implement the analysis pipelines. The focus is on automation and flexibility. Any task should be possible to run in an isolation on a small input. We often run into something unexpected in the corpus code and it helps a lot to be able to manually debug it without much setup overhead. To orchestrate the pipeline execution, we use GNU make. Each task is usually implemented either in R or bash. Tasks are run in parallel using GNU parallel [Tange et al., 2011]. The advantage of GNU parallel is that (1) it is simple to use, (2) it can run on one host or on multiple servers requiring only a password-less SSH setup,[12] and (3) it provides a comprehensive CSV file with the individual jobs' results. The RUNR package contains support for reading these results, allowing us can quickly figure out what went wrong. Since we execute real code from unknown sources, we use docker containers for sandboxing, protecting the execution environment from malicious or broken code. Another advantage of docker is that it facilitates the preparation of an artifact submission. Finally, for the actual data analysis, we use Rmarkdown.

2.3 Program Analysis

Program analysis helps us to answer questions about program behavior. In the case of static analysis, by looking at the program text. In the case of dynamic analysis, by actually running the program in a concrete execution environment and with a concrete input. This thesis relies on program analysis to get insights about language use in real-world code. In the case of R, we use dynamic analysis. In the case of Scala, we use static analysis. The domain of program analysis is relatively large. In this section, we discuss some work related to the studies included in this thesis.

[11] A popular analytics engine for big data processing, *cf.* `https://spark.apache.org/`

[12] With SSH connection sharing, there is very little overhead with spawning new remote jobs.

2.3.1 Automating Test Extraction

The literature can be split into work based on static techniques, dynamic techniques and a combination of both.

Static techniques use the program text as a starting point for driving test generation [Boyapati et al., 2002; Sen et al., 2005; Ernst et al., 2011]. The difficulty we face with R is its extreme dynamism (*cf.* Section 2.1). One approach that would be worth exploring, given that sound static analysis for R appears to be a non-starter, is some combination of static analysis and machine learning. For instance, MSeqGen uses data obtained by mining large code bases to drive test generation [Thummalapenta et al., 2009]. So far we have not gone down that road.

Dynamic approaches for generating unit tests often rely on some form of record and replay. Record and replay has been used to generate reproducible benchmarks from real-world workloads [Richards et al., 2011a], and, for example, capture objects that can be used as inputs of tests [Jaygarl et al., 2010]. Joshi and Orso describe the issues of capturing state for Java programs [Joshi and Orso, 2007]. Test carving [Elbaum et al., 2006] and factoring [Saff et al., 2005] have very similar goals to ours, namely to extract focused unit tests from larger system tests. These works mostly focus on capturing and mocking up a sufficiently large part of an object graph so that a test can be replayed. Rooney used similar approach to extract tests during an actual use of an application through instrumentation [Rooney, 2015]. While R has objects, their use is somewhat limited, they are typically self-contained, and capturing them in their entirety has worked well enough for now.

2.3.2 Type System Design

Dynamic programming languages such as Racket, JavaScript, PHP and Lua have been extended post factum with static type systems. In each case, the type system was carefully engineered to match the salient characteristics of its host language and to foster a particular programming style.

But what if the design of the type system is unclear? Andreasen et al. [2016] propose a promising 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.

2.3.3 Analyzing Scala Codebases

There have been efforts to study how Scala is used by practitioners. Tasharofi et al. [2013] looked at how often and why Scala developers mix the actor model with other models of concurrency. They analyzed 16 GitHub projects at the compiled byte-code

level with a custom tool. The choice of byte-code had some drawbacks. For example, their analysis could not detect indirect method invocations and thus they had to supplemented it with manual inspection. The same corpus is used by Koster [2015] to analyze different synchronization mechanisms used in Scala code. Despite using the same projects, he analyzed 80% more lines of code as the projects were updated to their latest commit. The increase was mostly due to Spark that grew from 12K to 104K lines of code in th two years that separated the studies. Unlike the previous study, they opted for source code analysis based on string matching. De Bleser et al. [2019] analyzed the tests of 164 Scala projects (1.7M lines of code) for a diffusion of test smells. They used a similar way of assembling a corpus relied on semantic data from the SCALAMETA. Villazón et al. [2019] build a language-agnostic dynamic analysis framework to to identify, among others, Java and Scala task-parallel workloads suitable for inclusion in a domain-specific benchmark suite. They analyze of 4K projects. We shall yet to evaluate if the presented framework could be used for our needs.

CHAPTER 3

Contribution

In this chapter we present the individual contribution from each of the enclosed publication. We extend the contribution summary from Section 1.2 with a context and motivation to introduce the problem, and description of the methodology used for the experiments including the design of the data analysis pipeline and corpus assembly. Each presentation is thus divided into three parts: (1) context and motivation, (2) methodology, and (3) results with key findings. We start with the work done in the context of R, followed by the study of implicits in Scala.

3.1 Unit Test Extraction

In this section we present an experiment with an automated test extraction from execution traces of client code in the R programming language. The work appeared in *Tests from Traces: Automated Unit Test Extraction for R* by *Filip Křikava and Jan Vitek*, published in *Proceedings of the 27th ACM International Symposium on Software Testing and Analysis (ISSTA)*, August 2018 [Křikava and Vitek, 2018a].

3.1.1 Context and motivation

This paper explores a relatively simple idea: *Can we effectively and efficiently extract test cases from program execution traces?* Our motivation is that if programmers do not write comprehensive unit test suites, then it may be possible for a tool to extract those for them, especially when the software to test is widely used in other projects. Our approach is as follows: (1) For each project and its reverse dependencies, gather all runnable artifacts, be they test cases or examples, that may exercise the target. (2) Run the artifacts in an environment where all project functions are instrumented and records execution traces. (3) From those traces, produce unit tests and, if possible, minimize them, keeping only the ones that increase code coverage.

We aim to answer the key question: *how well can automated trace-based unit test extraction actually work in practice?* The metrics of interest are related to the resulting coverage

of the target project and the costs of the whole process. Our goal is to extract black-box unit tests for software packages from a large body of client code. The benefit we are aiming for is to reduce the manual effort involved in constructing such regression testing suites and not necessarily finding new bugs. We try to answer this question in the scope of the R programming language. R is an interesting target because the CRAN packages, which are central to its ecosystem, have relatively low code coverage (on average 19%). However, at the same time, they are equipped with runnable code in the form of examples and vignettes that exercise their functionality.

3.1.2 Methodology

In the paper we report on implementation and empirical evaluation of GENTHAT, a tool that we have built for automated extraction of unit tests from R execution traces. First, we present the workings of the tool, followed by the design of the experiment that we used to evaluate it.

Genthat. An execution trace is a tuple $\langle f, v_1, \ldots, v_n, v, S \rangle$ recorded for each function call during the execution of a program, where f is a function identifier, v_i are values of the arguments, v is the return value, and S is the the current random number generator's seed. The seed is necessary for reproducing calls that involve random values.[1]

To automatically extract unit tests from execution traces, we essentially need to do four things:

1. Extract executable code from installed packages and turn them into R scripts. Each script is a self-contained runnable file.
2. Instrument functions in the target package to record execution traces run each extracted script.
3. Generate unit tests from the recorded traces. Concretely, we use `testthat` format[2] format as this is by far, the most popular unit testing framework in R.
4. Check the generated unit tests and discard any invalid and incorrect test. Invalid tests are those that fail to execute. Incorrect tests are those that do not yield the expected result. Optionally, we minimize the test suite. Minimization uses simple heuristics to discard tests that do not increase code coverage. Coverage being equal, tests that are, textually, smaller are preferred.

While seemingly simple, it takes a surprising amount of time to get right. There several engineering issues that are connected to the features of R, as mentioned in Section 2.1. We discuss them in detail in Section 3 of the paper (*cf.* Appendix A.1). Here we summarize the two main difficulties regarding execution tracing and unit test generation.

In a simpler language, all we would need to do is to capture function arguments, return values, and any global state the function may access on rely on. Not so in R. Lazy evaluation complicates matters as arguments are passed by *promises*. A promise is

[1] Being a language for statisticians, random number generators are omnipresent.
[2] *cf.* https://testthat.r-lib.org/

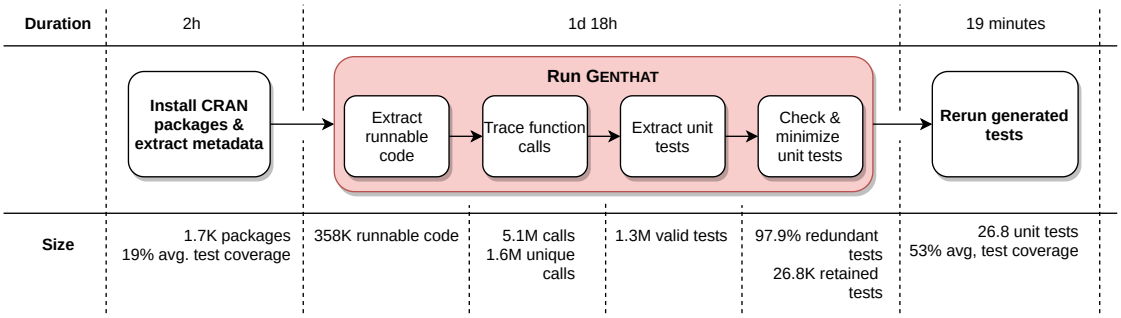


Figure 3.1: Overview of the pipeline. GENTHAT extracts scripts, traces their execution, and generates unit tests. Unit tests are checked for correctness and validity, and finally tests are minimized. The experiments were carried out on two virtual nodes (each 60GB of RAM, 16 CPU at 2.2GHz and 1TB virtual drive)

a closure that is evaluated at most once to yield a result. Thus, at function call, one cannot directly record values as they are not yet evaluated. Doing so would force promises that might not be evaluated in the current frame. It may also alter the behavior of the program since promises can have side effects. Moreover, R's reflective capabilities dictate that the generated expression must be as close to the original call as possible. Any syntactic change may be observable by user code. The generated test thus attempts to retain the structure of the client source code as much as possible.

To generate unit tests out of the recorded trace, we need to write out arbitrary R values into a source code file. The `deparse` function turns some values into a character string. Unfortunately, it handles only a subset of the 25 different R data types. Since any of those data types can show up as an argument to a function call, we had to implement own deparsing mechanism to support them all. In general, we strive to output textual forms of arguments because they can be inspected and modified by developers. Nevertheless, there are values for which it is either impractical (*e.g.*, large vectors or matrices) or not possible to turn them back into source code. For those, we fallback to built-in binary serialization.

Pipeline. Figure 3.1 presents an overview of the pipeline used for the experiment. We start by installing all of the selected CRAN packages and extracting metadata, including test code coverage. Next, we run GENTHAT. Finally, we rerun the extracted tests. We do two runs, one to measure how much time it takes to run and second in which we measure code coverage.

Test extraction can fail. In the paper, we provide a detailed analysis of the reasons (*cf.* 4.4 in Section 3.1), but at a high-level, the failures can occur during: (1) tracing because the instrumentation perturbs the behavior of the program, (2) generation because some value could not be serialized, (3) validation because deserialization fails, or (4) correctness checking because the test was non-deterministic or relied on an external state that was not captured.

Corpus. For these experiments, we selected 1,700 packages from CRAN. We picked the 100 most downloaded packages from RStudio CRAN mirror[3], including their dependencies, and then 1000 randomly selected CRAN packages, again including their dependencies. This added up to some 1,726 packages. The motivation for this choice is to have some well established and popular packages along with a representative sample of CRAN. From the 1.7K packages, some 1,524 ran successfully. The remaining failed either because of a timeout (5 hours during tracing), a runtime error, or failure to compute code coverage. The packages amounted to 1.7M lines of R code. There were 158K lines of examples, 32K lines of code extracted from vignettes, and 163K lines of code in tests.

3.1.3 Results

A detailed discussion about the results is provided in Section 4 of the paper (*cf.* Appendix A.1). Here we provide a summary of GENTHAT evaluation. We were primarily interested in finding out (1) how much we can improve test coverage by extracting tests from documentation and reverse dependencies, (2) how efficient such extraction can be, and (3) what proportions of the functions calls can be turned into test cases and how large the resulting test suites would become.

Coverage. Figure 3.2 compares the code coverage obtained from the existing package tests and the coverage obtained from GENTHAT generated unit tests. For many packages, the provided tests give relatively low coverage. GENTHAT is able to increase the coverage from an average of 19% to 53% per package. This is a significant improvement.

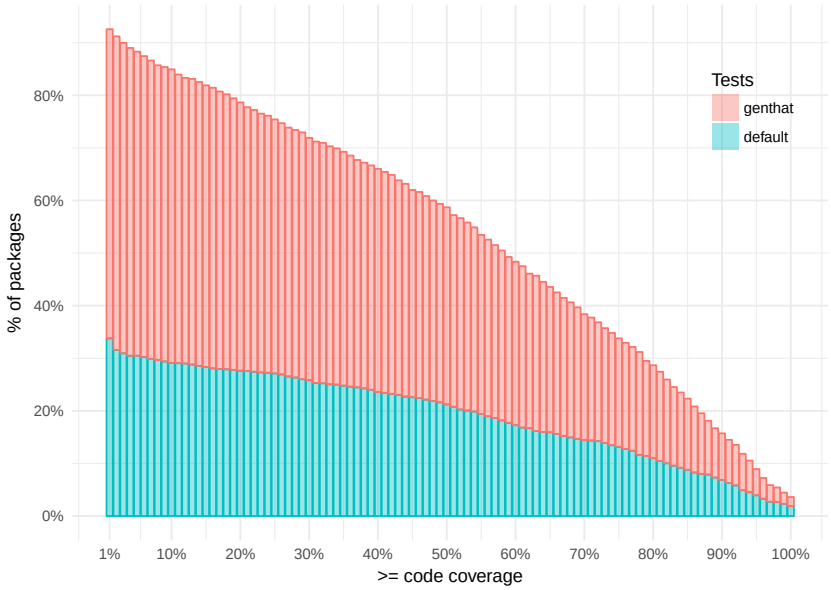


Figure 3.2: Code coverage from package tests and GENTHAT extracted unit tests. Each bar shows the ratio of packages that have at least that code coverage.

[3]Used data for June 2017 from http://cran-logs.rstudio.com.

Performance. Tracing clearly has a significant overhead. The overall runtime is almost two days on a two-node virtual cluster. Per package, the median time is 60 seconds. Given that test generation is a design-time activity, this is likely acceptable. The main portion of the time is spent in running the generated tests, which are numerous. On average, each package yields 1,000 tests. They all have to be run to determine validity and correctness, as well as test minimization to determine whether they increase code coverage. On the other hand, testing with the minimized tests run fast. We can process all the packages in 19 minutes. Thanks to test minimization, this is 1.9 times faster than the time it would take to run all of the original examples, tests, and vignettes.

Accuracy. Running the executable code extracted from the 1.5K corpus packages resulted in 5.3M execution traces. Out of them, only 1.6M unique calls—*i.e.*, calls with distinct arguments and return values. 93% of these traces were turned into using tests. 86% of the generated tests were valid and correct. From the total of 1.3M correct tests, 97.9% were redundant—*i.e.*, not increasing code coverage and therefore discarded. Finally, some 26,838 tests were retained.

3.2 Towards a Type System for R

In this section we present the experiment with a design of a type system for the R programming language. The work appeared in *Designing Types for R, Empirically* by *Alexi Turcotte, Aviral Goel, Filip Křikava and Jan Vitek* published in *Proceedings of the ACM Programming Languages 4, OOPSLA*, Article 181, November 2020 [Turcotte et al., 2020c].

3.2.1 Context and motivation

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 salient ones, the lack of type annotations, the ability to use syntactic shortcuts, and widespread conversion between data types. While these choices have decreased the barrier to entry—many data science educational programs do not teach R itself but simply introduce some of its essential libraries—they also allow for errors to go undetected.

Retrofitting a type system to the R language would increase our assurance in the result of data analysis, but this requires facing two challenges. First, it is unclear what would be the *right* type system for a language as baroque as R. Second, but just as crucially, designing a type system that will be adopted would require overcoming some prejudices and educating large numbers of users.

The goal of this paper is to gather data that can be used as input to the process of designing a type system for R. The long-term goal is to retrofit a type system into R. In this work we start by looking at function signatures.

3.2.2 Methodology

We attempt to address ascribing function signatures using an iterative process: First, we design a simple type language that matches the R data types. Next, we record function invocations for a corpus of widely used libraries. We do this by executing the runnable code extracted from these packages. As a next step, we use the traces to infer function signatures. These signatures are then turned into contracts injected into the corresponding functions and checked whether they well describe the function arguments and return values. We check this by running code from reverse dependencies as well as code from Kaggle. Finally, we use the feedback to improve the type language design. This allows us to see how far one can get with a simple type language and identify our design limitations.

Pipeline. To support this process, we have built tooling to automate the extraction of raw type signatures from execution traces, infer type signatures from a set of raw types, and validate the inferred signatures by means of contracts. These tools are used in an automated pipeline (*cf.* Figure 3.3). It includes the following steps:

1. Download and install all available CRAN packages.
2. Extract package metadata including code coverage and reverse code coverage—*i.e.*, the code coverage obtained from running code from dependent packages.
3. Running dynamic analysis to trace function invocation and record information of types of function arguments and return values.
4. Infer function type signatures from gathered type information.
5. Validate inferred signatures by running code from reverse dependencies.

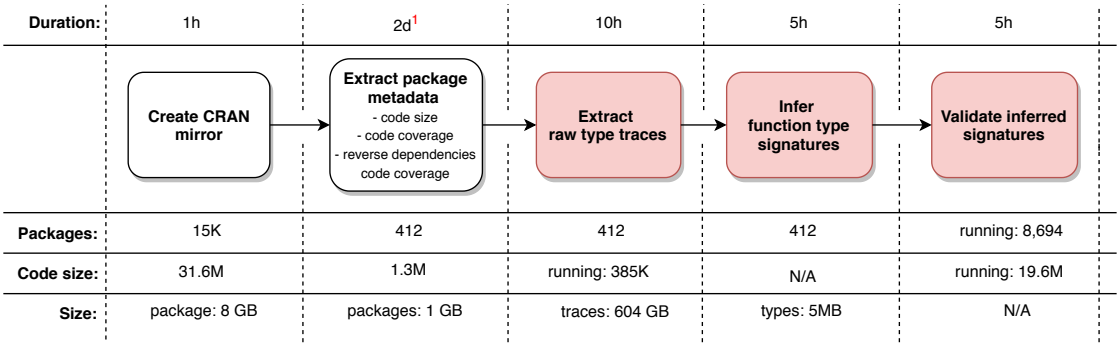


Figure 3.3: The Analysis Pipeline; [1] metadata has to be extracted for *all* CRAN packages. The experiments were carried on single node Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM.

Corpus. We have selected 412 CRAN packages consisting of 760.6K lines of R code and 534.4K lines of native code. We used two criteria for including a package into the corpus: (1) the package must have 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 allow 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 412 selected packages contain 385.8K lines of runnable code. There are 18.8K reverse dependencies with 11.2M runnable lines of code resulting in 45.9% coverage (on average) of the corpus packages. To represent end-user code in the corpus, we turned to Kaggle using 792 kernels with 33.7K lines of R code.

Type language. Figure 3.4 resents the type language. Given the language's peculiarities (cf. Section 2.1), several design choices had to be reviewed and evaluated. Here we highlight the essentials. We include types that cover all primitive vectors, environments, lists, and the distinguished null types, inhabited by the single `NULL` value. To capture common uses of NULL, the type language has a nullable type, written $?\ T$ and representing either values of type T or NULL. We also add basic support for classes—*i.e.*, any non-null R value with a `class` attribute. Despite that R does not have scalars, there are cases where functions expect scalar values, so we add a distinction between vectors (*e.g.*, `int` []) and scalars (*e.g.*, `int`). Each primitive type has its specific `NA`. Many built-in functions, especially those implemented in C or Fortran, do not support `NA` values. It is thus advantageous to distinguish between vectors that can and cannot contain missing values. Finally, We support untagged unions of types written $T_1 \mid \ldots \mid T_n$ and functions signatures the form $\langle A_1, \ldots, A_n \rangle \to T$ where each A_i argument is either a type T or dots $(\ldots)$, a variable-length argument list. A single function's signature can be the disjunction of a number of individual signatures. Overall, the choice of type language follows the structure of values. The presence of `NA`-free data types and scalars are two choices that must be validated in practice.

T	::=	`any`	top type	A	::=	T	arguments
	|	`null`	null type		|	$\ldots$	dots
	|	`env`	environment type	V	::=	S []	vector types
	|	S	scalar type		|	^S []	na vector types
	|	V	vector type	S	::=	`int`	integer
	|	$T \mid T$	union type		|	`chr`	character
	|	$?\ T$	nullable type		|	`dbl`	double
	|	$\langle A_1, \ldots\ A_n \rangle \to T$	function type		|	`lgl`	logical
	|	`list`$\langle T \rangle$	list type		|	`clx`	complex
	|	`class`$\langle ID_1, \ldots\ ID_n \rangle$	class type		|	`raw`	raw

Figure 3.4: The R type language

R does not support the notion of subtyping between values, but we include a few simple rules to capture widespread coercion, including the new rules created by our type language (*cf.* Figure 3.5).

Type tracing and type inference. We use dynamic analysis for extracting raw type traces. Similar to execution traces in GENTHAT, a type trace is a tuple $\langle f, t_1, \ldots, t_n, t \rangle$ for each function call during the execution of a program, where f is a function identifier, t_i

$$
\begin{array}{rcl}
S\,[\,] & <: & \hat{S}\,[\,] \\
T & <: & \hat{?}T \\
\texttt{list}\langle T\rangle & <: & \texttt{list}\langle T'\rangle \ \textit{iff}\ T <: T' \\
S & <: & S\,[\,] \\
\texttt{lgl} & <: & \texttt{int} \\
\texttt{int} & <: & \texttt{dbl} \\
\texttt{dbl} & <: & \texttt{clx}
\end{array}
$$

Figure 3.5: Subtyping rules

$$
\begin{array}{rcl}
T \mid_{\dot{}} T & \Rightarrow & T \\
T \mid_{\dot{}} T' & \Rightarrow & T \ \textit{iff}\ T' <: T \\
\texttt{list}\langle T\rangle \mid_{\dot{}} \texttt{list}\langle T'\rangle & \Rightarrow & \texttt{list}\langle T\rangle \ \textit{iff}\ T' <: T \\
\texttt{null} \mid_{\dot{}} S_1[\,] \mid_{\dot{}} \ldots S_n[\,] & \Rightarrow & \hat{S}_1[\,] \mid_{\dot{}} \ldots \hat{S}_n[\,]
\end{array}
$$

Figure 3.6: Simplification rules

are types of the arguments and t is a type of the return value. Similarly to GENTHAT, extra care has to be done for promises. To deal with unevaluated arguments, we make an initial guess for each argument at function entry and update the recorded type if the promise is forced.

These traces will be combined into an overall arrow type for the function, and in order to keep these signatures compact, we unify the traces into a single top-level arrow with unions at each argument position. Thus, the shape of function signatures will be:

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

In other words, we take the union of the types occurring at individual argument positions rather than a union of function types. Furthermore, we apply some transformation on the types to keep the size of types in check (*cf.* Figure 3.6).

Checking type signatures One can validate a function's type signature by checking that it is respected in all programs that call the function. For this, we developed CONTRACTR, an R package that decorates functions with assertions. We use it to insert the type checking code around functions.

The injected contracts check arguments with a simple tag check when possible. Some properties require traversing data structures, such as the absence of `NA`. For union types, multiple checks may be needed, at worst one per member of the union. To retain the non-strict semantics of R, the expression held in a 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.

3.2.3 Results

A detailed discussion about the results is provided in Section 6 of the paper (*cf.* Appendix A.2). Here we provide a summary of the evaluation. We were primarily interested in finding out (1) how expressive the type language is, (2) how robust it is, and (3) how useful the inferred signatures are. Figure 3.7 gives an example of how the inferred signatures look like.

Expressiveness. The first part of our evaluation attempts to shed light on how good a fit our proposed type language is with respect to common programming patterns oc-

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**[]

Figure 3.7: Selected type signatures

curring in widely used R libraries.

- Nearly 80% of functions are either monomorphic or have only one single polymorphic argument. Monomorphic in this context means that the type is not relying on `any` or including a union.
- Most functions do not require a union at all (83.1% of arguments do not have a union), and only 2.5% of positions have unions with more than three members.
- The 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.
- 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.
- 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.
- There are just 1,705 argument positions (1.5%) that take higher-order functions. Given the small number of occurrences, it is not worth complicating the inferred types with a complete signature for these functions.

3.2.4 Robustness

To measure the inferred types' robustness, we check the signature contracts by running the extracted code from the reverse dependencies of the corpus's packages. In total, we ran extracted code from 8.6K unique packages and recorded 98M total assertions. Overall, we found that only less than 2% of contract assertions failed. Overall, these numbers are promising and suggest that the type signatures are indeed robust.

Usefulness. One way to assess our type checking framework's usefulness is to see how many existing user-defined type checks can be replaced by CONTRACTR. To mea-

sure this, we extracted all calls to `stopifnot`[4] and `assertthat`[5] assertions, and checked which could be either simplified or be completely replaced by CONTRACTR.

Out of the 412 packages, 153 use runtime assertions. Altogether, there are 1,995 assertions in 1,264 functions. Among these, CONTRACTR can replace 1,005 (50.4%) assertion calls across 114 packages, and 688 functions. Furthermore, an additional 218 asserts in 125 packages and 859 functions could have been simplified.

3.3 Study of Implicits in Scala

In this section we present the study of implicits in the Scala programming language. This work appeared in *Scala Implicits are Everywhere: A Large-scale Study of the Use of Scala Implicits in the Wild* by *Filip Křikava, Heather Miller and Jan Vitek* published in *Proceedings of the ACM Programming Languages 3, OOPSLA*, Article 163, October 2019 [Křikava et al., 2019].

3.3.1 Context and motivation

Language ergonomics is essential. Language designers try hard to find ways to reduce the friction that users experience when expressing programming tasks [Turon, 2017]. There is always a trade-off between conciseness and readability. The ergonomics is not necessarily about less typing but rather about reducing the boilerplate code and the distraction of expressing the implied. One way of doing this is to rely on a compiler and its knowledge and understanding of the code to fill the "boring parts." For example, in functional programming, we often have to pass around a shared context, which can get tedious and error-prone. A compiler could instead provide such an argument implicitly. This idea of *implicit parameters* has been first explored by Lewis et al. [2000] in Haskell and later popularized by Scala [Odersky et al., 2006] and by many other languages, *e.g.*, Agda [Norell, 2007], Coq [Sozeau and Oury, 2008], Idris [Brady, 2013], OCaml [White et al., 2015], and Flix [Madsen and Lhoták, 2018]. Even Google Torque[6], a language for developing the JavaScript V8 engine, features implicit parameters.

Next to implicit parameters, Scala also contains *implicit conversion*, which lets the compiler automatically adapt data structures to the proper interface without the need for explicit calls to constructors. Together, Scala *implicits* offload the task of selecting and passing arguments to functions and converting between types to the compiler. This allows one to implement language features outside of the compiler [Miller et al., 2013] and write code with less boilerplate [Haoyi, 2016]. They have changed the way Scala is used. Oliveira C. d. S. et al. [2010] showed how to use implicit parameters to implement type classes [Wadler and Blott, 1989], which gave rise to complete ecosystems of libraries for functional programming[7]. They have been used for embedding DSL in

[4] R assertion built-in.
[5] The most popular assertion library in R *cf.* `https://github.com/hadley/assertthat`
[6] *cf.* `https://v8.dev/docs/torque`
[7] *cf.* typelevel.scala (`https://typelevel.org/`) and ZIO (`https://zio.dev/`)

Scala, establish or pass context, dependency injection, modeling capabilities, configurations, and many other forms of contextual abstractions [Odersky et al., 2017; Miller et al., 2014]. As Martin Odersky puts it, *"If there's one feature that makes Scala "Scala", I would pick implicits."* [Odersky, 2017]

```
// defines extension methods for String instances
implicit class StringEx(that: String) {
  def enEspanol(implicit srv: Translator): String = srv.translate(that)
}
// defines a new Translator instance available in the implicit scope
implicit val traslator: Translator = ...
```

Listing 3.1: Example of Scala implicits

Example. Let's consider the following code snippet: `"Just like magic!".enEspanol` Without additional context, one would expect the code not to compile as the `String` class does not have `enEspanol` method. In Scala, however, if the compiler can find a conversion[8] between a string and an instance of a class that has the required method (which resolves the type error), that conversion will be inserted silently by the compiler and, at runtime, the method will be invoked to return a value, perhaps `"Como por arte de magia!"`. The method will be defined with an implicit parameter to refer to a service that can do the actual translation. A possible implementation is shown in Listing 3.1. With these definitions imported into the scope of the original code snippet, the compiler will rewrite the call to `new StringEx("Just like magic!").enEspanol(translator)`.

The problem with implicits. While powerful, implicits are not without flaws:

- *Anti-patterns.* They can be easily misused, leading to a code that is hard to manage [Odersky, 2017]. A widely discussed anti-pattern is the conversion between types in unrelated parts of the type hierarchy. The perceived danger is that any type can be automatically coerced to a random type unexpectedly. Another anti-pattern is conversions that go both ways. Since conversions are not visible, it is difficult to reason about types at a given call site as some unexpected conversion could have happened. An example is the, now deprecated, Java collection conversion. As they were often imported together using a wildcard import, it was easy to mistakenly invoke a Java method on a Scala collection and vice-versa, silently converting the underlying data structures and possibly changing semantics[9].
- *Complexity.* Complex use of implicits can lead to confusing scenarios or difficult-to-understand code. Understanding implicit-heavy code can place an unreasonable

[8] Implicit conversions also appear in languages such as C++ or C#. However, the difference is that their conversions are typically defined in the class participating in the conversion. In contrast, in Scala, the implicit conversions can be defined in types unrelated to the conversion types. This allows programmers to import conversion selectivelly. For instance, define an implicit conversion between `String` an `Int`.

[9] The notion of equality in Java collections is different from Scala collections (reference vs. element equality)

burden on programmers. A good example is the community backlash [Marshall, 2009] following the introduction of the Scala 2.8 Collections library [Odersky and Moors, 2009], a design which made heavy use of implicits in an effort to reduce code duplication. The design caused a proliferation of complex method signatures across common data types throughout the Scala standard library. This eventually led to the addition of *use-cases*[10] into the Scala documentation tool to simplify complex method signature.

– *Overhead.* Implicits have been observed to affect compilation-time performance, sometimes significantly. For example, a popular Scala project reported a three order-of-magnitude speed-up when developers realized that an implicit conversion was silently converting Scala collections to Java collections only to perform a single operation that should have been done on the original object.[11] Another project reported a 56-line file taking 5 seconds to compile because of implicit resolution. A one-line fix changing the scope of one implicit definition improved compile time to a tenth of second [Torreborre, 2017].

This work. Most of the above issues were reported in the form of anecdotal evidence. This paper aims to provide empirical evidence about the use of implicits in the Scala ecosystem. To document, for language designers and software engineers, how this feature is really used in the wild, using a large-scale corpus of real-world programs. We provide data on how they are used in popular projects engineered by expert programmers as well as in projects that are likely more representative of how the majority of developers use the language. This paper is both a retrospective on the result of introducing this feature into the wild, as well as a means to inform designers of future languages of how people use and misuse implicits.

3.3.2 Methodology

To understand the use of implicits across the Scala ecosystem, we have built an open-source and reusable pipeline to automate the analysis of large Scala codebases, compute statistics, and visualize results.

Pipeline. Figure 3.8 gives an overview of the pipeline. It includes the following steps:

1. Download projects hosted on GitHub.
2. Gather necessary metadata and, in particular, infer the build system each project uses.
3. Discard projects that do not meet the technical requirements of the analysis tools.[12]
4. Filter out duplicate and uninteresting projects.
5. Compile the corpus and generate semantic information.
6. Extract implicit usage from the semantic database.
7. Analyze the data.

[10] *cf.* https://docs.scala-lang.org/overviews/scaladoc/for-library-authors.html
[11] *cf.* https://github.com/mesosphere/marathon/commit/fbf7f29468bda2ec29b7fbf80b6864f46a825b7a.
[12] We only support SBT based projects with Scala version >= 2.11

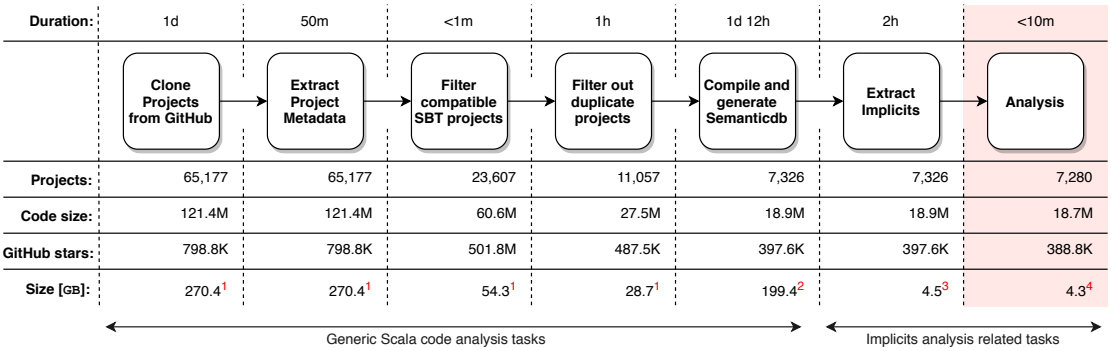


Figure 3.8: Scala Analysis pipeline. (1) is the size of source code, (2) is the size of source plus compiled code and generated SEMANTICDB, (3) is the size of the extracted implicits data model, (4) is the size of exported CSV files. The code size includes tests. The experiments were carried on single node Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM.

The pipeline is reusable for other semantic analyses on Scala codebases, as only the last two steps relate specifically to implicits. At the end of the *Compile and generate* SEMANTICDB task, the corpus contains built projects with extracted metadata and SEMANTICDB files. These files contain low-level syntactic information as well as semantic information about a compilation unit. Among others, it contains a list of symbols with their types and a list of synthetic calls injected by the compiler. SEMANTICDB is generated using the SCALAMETA. The low-level data from SEMANTICDB are not suitable for higher-level queries about the use of implicits. To this end, we have developed a static analyzer that transforms the semantic data into our model that is easy to query. It works in two steps: first, it resolves all implicit symbols and then extracts implicit function applications from the synthetic calls. The resulting model can be queried using the standard Scala collection API. Listing 3.2 shows how to list all implicit declarations of the `Translator` class (*cf.* Listing 3.1) and the related call sites that use.

```
val declarations = proj.declarations.filter(dcl =>
  dcl.isImplicit && dcl.isVal && dcl.returnType.isKindOf("Translator"))
val callsites =
  proj.implicitCallsites.filter(cs =>
    cs.implicitArguments.exists(arg => declarations contains arg))
```

Listing 3.2: Example of a query for implicits use

Corpus. We acquired and processed a corpus of 7,280 projects from GitHub with over 18,713,247 non-empty lines of Scala code. The corpus was obtained from publicly available projects listed in the GHTorrent database [Gousios, 2013] and Scaladex[13]. We started with 65,177 non-empty, non-fork projects with 121.4M lines of code. Out of that, we fil-

[13] A package index of Scala projects published in Maven Central and Bintray, *cf.* https://index.scala-lang.org/.

tered out 41.5K incompatible and 12.5K duplicate / uninteresting projects. Since GitHub is not a curated repository, it was essential to address the duplicates (*cf.* Section 2.2).

3.3.3 Results

A detailed discussion about the results is provided in Section 5 of the paper (*cf.* Appendix A.3). Here we will summarize the critical findings organized in the four main categories: usage overview, patterns, complexity, and compilation-time overhead.

- *Usage overview.* Scala implicits are everywhere! Figure 3.9 shows the high-level overview of the implicit usage in the corpus. Out of the 7,280 analyzed projects (18.7M lines of code), 7,148 (98.2%) have at least one implicit call site. From over 29.6M call sites in the corpus (explicit and implicit combined), 8.1M are call sites involving implicits. Most of these calls are related to the use of implicit parameters (60.3%). There are roughly twice as many implicit call sites (38%) in tests than in the rest of the code (17.1%). That is not surprising because the most popular testing frameworks heavily rely on implicits. Across the project categories, the median is 23.4%—*i.e.*, *one out of every four call sites involves implicits*. There are 5,694 (78.2%) projects that together define 370.7K implicit declarations.
- *Patterns and anti-patterns.* We have looked at the usage of the six main patterns: late trait implementation (allowing a class to implement an interface after the class has been defined), extension methods (adding methods into existing classes), type classes, extension syntax methods (a combination of type classes with extension methods), type proofs and contexts; and two main anti-patterns: unrelated and bidirectional conversions. The majority of projects use type classes and extension methods. As expected, they are mostly defined in libraries.

  About 6.1K projects use unrelated conversions, but only 552 using unrelated conversions between primitive types. They are present in all categories, but the majority comes from libraries where they are used as building blocks for DSLs. A very few (81 in 47 projects) convert just between primitive types. In 1.9K projects, we have seen the use of bidirectional conversion. From a manual inspection of bi-directional conversion, we find that they are mainly used in libraries that provide both Java and Scala API. This suggests that most implicit conversions are safe.
- *Complexity.* Most of the call sites that involve implicits receive between 0-2 implicit arguments. However, the distribution has a long tail, and the project that rely on type-level programming contain implicit-heavy call sites. With the increasing popularity of libraries that use implicit type-class derivation for algebraic data types, the programmers are likely to deal with more complicated calls. To help navigate this complexity, the Scala plugin for Intellij IDEA has a feature that can show implicit hints, including implicit resolution in the code editor. This effectively reveals the injected code making it an indispensable tool for debugging. However, turning the implicit hints on severely hinder's the editor performance creating a significant lag when working with implicits-heavy files.

– *Overhead*. We looked at the compilation times of all 488 projects (2.8M lines of code) that use the `shapeless` library and compared them to 1.9K other projects that use Scala version for, which we could get detailed compilation statistics[14]. This `shapeless` library provides the most common approach for implicit type-class derivation [Cantero, 2018]. The data confirm the hypothesis that the cost of compilation increases with the density of implicits, and the use of type classes further reduces compilation speed.

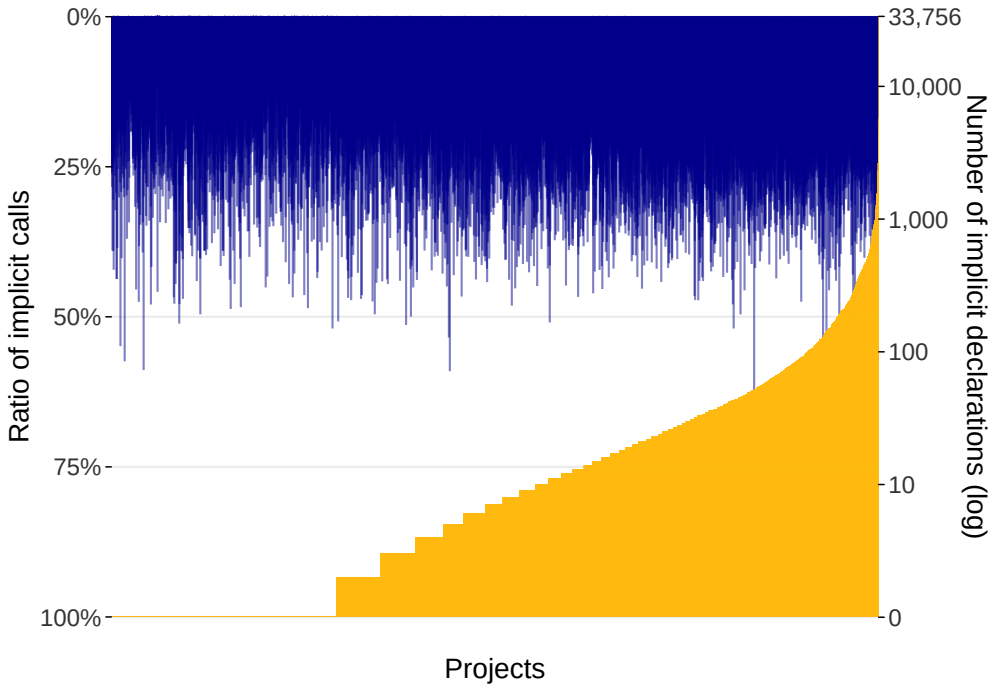


Figure 3.9: Overview of the implicits in the corpus. The top of the graph shows the ratio of call sites in each project that involves implicit resolution. The bottom shows the number of implicit definitions in each project.

[14]The `-Ystatistics:typer` compiler flag was introduced in Scala 2.12.4.

CHAPTER 4

Conclusions

Democratizing code sharing through large code-hosting websites opens whole new opportunities in understanding and improving programming languages. Using program analysis, we can get empirical evidence about how languages and libraries are used in real-world projects. These insights can then help us better evaluate the changes needed to grow languages to better adapt to their users' needs.

In this thesis, we have presented three large-scale program analyses. They all seek answers to questions to understand better the use of the R and Scala programming languages:

- In *Tests from Traces: Automated Unit Test Extraction for R*, the question we aim to answer is: *how well can automated trace-based unit test extraction actually work in practice?* The metrics of interest are the quality of the extracted tests, their coverage of the target project, and the costs of the whole process. To answer it, we have built Genthat, a tool for the automated extraction of unit tests for the R language. Our evaluation of this tool on a large corpus of packages suggests that it can significantly improve coverage. On average, the default tests that come with the packages cover only 19%, while with tests extracted by GENTHAT, the coverage increases to an average of 53%.
- In *Designing Types for R, Empirically*, the question was about *what expressive power do we need to ascribe types to R function?* The longer-term goal is to retrofit a type system into R, but in this work, we only look at function signatures. To answer the question, we have designed a simple type language focusing on the most widely used features of R. We evaluated the design by inferring and subsequently checking function signatures on several widely used R packages. 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, less than 2% of contracts generated from the inferred function signatures failed at runtime. Moreover, using the type language would eliminate or otherwise simplify 61% of existing type checks and assertions in user code.

– In *Scala Implicits are Everywhere* we asked *how Scala implicits are used in the wild?* To answer this question, we have analyzed all open-sourced Scala projects available on GitHub for which we could extract semanatic information. The results showed that Scala developers embraced them. Almost all projects use at least one implicit parameter or implicit conversion, and over 78.2% define them. Every fourth call site involves the use of implicits. Most of the implicit call sites are simple, but the complexity increases with libraries that rely on implicit type-class derivation. The same libraries also increase the compilation time.

Program analysis of large code repositories is hard. In part, this is because scaling program analysis is hard. A linear increase in the analyzed code size results in a non-linear increase in the analysis's complexity. But part of the challenge is that substantial code only comes in the context of realistic programming language that has been around for a while. Working with real code invariably entails infrastructure building, which requires substantial engineering effort.

In the case of R, this was particularly painful as we had to develop the complete analysis toolchain. However, now we can finally amortize the engineering cost and use the tools for a number of future work: Continue in the endeavor of retrofitting a type system into R. Extend GENTHAT with symbolic execution and value fuzzing to generate more tests and extend the code coverage of the R package. Study the use of eval in R and work on tooling that could remove it. Finally, we would like to put the insights into real use by feeding them into Ř, an alternative just-in-time compiler for R developed in our research group [Flückiger et al., 2020].

APPENDIX A

Papers

In this appendix, we enclose the copies of the papers presented in this thesis. They are in their published version, including the ACM badges[1] received from the artifact evaluation from each corresponding conference.

[1] Details about ACM artifact reviews and badging can be found at ACM website, *cf.* https://www.acm.org/publications/policies/artifact-review-badging

A.1 Tests from Traces: Automated Unit Test Extraction for R

Tests from Traces: Automated Unit Test Extraction for R by *Filip Křikava and Jan Vitek*, published in *Proceedings of the 27th ACM International Symposium on Software Testing and Analysis (ISSTA)*, August 2018 [Křikava and Vitek, 2018a].

Tests from Traces: Automated Unit Test Extraction for R

Filip Křikava
Czech Technical University
Czech Republic

Jan Vitek
Northeastern University and CTU
USA

ABSTRACT

Unit tests are labor-intensive to write and maintain. This paper looks into how well unit tests for a target software package can be extracted from the execution traces of client code. Our objective is to reduce the effort involved in creating test suites while minimizing the number and size of individual tests, and maximizing coverage. To evaluate the viability of our approach, we select a challenging target for automated test extraction, namely R, a programming language that is popular for data science applications. The challenges presented by R are its extreme dynamism, coerciveness, and lack of types. This combination decrease the efficacy of traditional test extraction techniques. We present Genthat, a tool developed over the last couple of years to non-invasively record execution traces of R programs and extract unit tests from those traces. We have carried out an evaluation on 1,545 packages comprising 1.7M lines of code. The tests extracted by Genthat improved code coverage from the original rather low value of 267,496 lines to 700,918 lines. The running time of the generated tests is 1.9 times faster than the code they came from.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Test extraction, Program tracing, R

ACM Reference Format:
Filip Křikava and Jan Vitek. 2018. Tests from Traces: Automated Unit Test Extraction for R. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3213846.3213863

1 INTRODUCTION

Testing is an integral part of good software engineering practices. Test-driven development is routinely taught in Computer Science programs, yet a cursory inspection of projects on GitHub suggests that the presence of test suites cannot be taken for granted and even when tests are available they do not always provide sufficient coverage or granularity needed to easily pinpoint the source of errors.

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213863

This paper explores a rather simple idea, namely, can we *effectively and efficiently extract test cases from program execution traces*? Our motivation is that if programmers do not write comprehensive unit test suites, then it may be possible for a tool to extract those for them, especially when the software to test is widely used in other projects. Our approach is as follows: For each project and its reverse dependencies, gather all runnable artifacts, be they test cases or examples, that may exercise the target, run the artifacts in an environment that records execution traces, and from those traces produce unit tests and, if possible, minimize them, keeping only the ones that increase code coverage.

The key question we aim to answer is: *how well can automated trace-based unit test extraction actually work in practice?* The metrics of interest are related to the quality of the extracted tests, their coverage of the target project, as well as the costs of the whole process. To answer this we have to pick an actual programming language and its associated software ecosystem. Any combination of language and ecosystem is likely to have its quirks, we structure the paper so as to identify those language specific features.

Concretely, this paper reports on our implementation and empirical evaluation of Genthat, a tool for automated extraction of unit tests from traces for the R statistical programming language [11]. The R software ecosystem is organized around a curated open source software repository named CRAN. For our purposes, we randomly select about 12% of the packages hosted on CRAN, or 1,545 software libraries. These packages amount to approximately 1.7M lines of code stripped of comments and empty lines. The maintainers of CRAN enforce the presence of so-called vignettes, these are documentation with runnable examples, for all hosted libraries. Some libraries come equipped with their own test cases. These are typically coarse grained scripts whose output is compared for textual equality. Our aim with Genthat is to help R developers extract unit tests that can easily pinpoint the source of problems, are reasonably small, provide good coverage, and execute quickly. Furthermore, we want to help developers to automatically capture common usage patterns of their code in the wild. In the R code hosted on CRAN, most of the code coverage comes from examples and vignettes, and very little is already in the form of tests. In the corpus of 1,545 packages we selected for this work, tests provide only an average of 19% coverage, whereas when examples and vignettes are executed coverage is boosted to 68%.

As of this writing, we are not aware of any other tool for automatically extracting test cases for R. This likely due to the limited interest that data analysis languages have garnered in our community, and also due to features of R that make it a challenging target for tooling. The language is extremely dynamic: it has no type annotations to structure code, each and every operation can be redefined during execution, values are automatically coerced from type to type, arguments of functions are lazily evaluated and can be coerced back to source expressions, values are modified using a

copy-on-write discipline most of the time, and reflective operations allow programs to manipulate most aspect of a program's execution state [9]. This combination of features has allowed developers to build complex constructions, such as support for object-oriented programming, on top of the core language. Furthermore, large swaths of the system are written in C code and may break any reasonable invariants one may hope for.

The contributions of this paper are the description of a tool, GENTHAT, for automatically extracting unit tests for R, as well as an empirical evaluation of that tool that demonstrates that for a large corpus, 1,545 packages, it is possible to significantly improve code coverage. On average, the default tests that come with the packages cover only 19%. After deploying GENTHAT we are able to increase the coverage to 53%. This increase mostly comes from extracting test cases from all the available executable artifacts in the package and the artifacts from packages that depend on this package. GENTHAT is surprisingly accurate, it can reproduce 80% of the calls executed by the scripts and it is also able to greatly reduce the number and size of test cases that are retained in the extracted suite, running 1.9 times faster than package examples, tests and vignettes combined with only 15% less code coverage (53% vs 68%). The reduction in coverage comes from limitations in what code can be turned into tests.

Artifact. The code of our tools, the analysis scripts used to generate the reports and sample data are available in the validated artifact accompanying this paper[1].

2 BACKGROUND AND RELATED WORK

This section starts with a brief overview of the relevant characteristics of the R programming language, as well as of its ecosystem, before discussing previous work on automated test extraction and its application to R.

2.1 The Language

The R programming language is a challenging language for tooling. The relevant features are the following:

- R does not have type annotations or a static type system. This means there is nothing to suggest what the expected arguments or return values of a function could be, and thus there is little to guide test generation.
- Symbols such as `+` and `(` can be redefined during execution. This means that every operation performed in a program depends on the state of the system, and no function call has a fixed semantics. This holds even for control flow operations such as loops and conditionals. While redefinitions are not frequent, they do occur and tools must handle them.
- Built-in types are automatically and silently coerced from more specific types to more general types when deemed appropriate.
- R is fully reflective, it is possible to inspect any part of a computation (e.g. the code, the stack or the heap) programmatically, moreover almost all aspects of the state of a computation can be modified (e.g. variables can be deleted from an environment and injected in another).
- All expressions are evaluated by-need, thus the call `f(a+b)` contains three delayed sub-expressions, one for each variable and one for the call to plus. This means that R does not pass values to functions but rather passes unevaluated promises (the order of evaluation of promises is part of the semantics as they can have side effects). These promises can also be turned back into code by reflection.
- Most values are vectors or lists. Values can be annotated by key-value pairs. These annotations, coupled with reflection, are the basic building blocks for many advanced features of R. An example of this are the four different object systems that use annotations to express classes and other attributes.
- R has a copy-on-write semantics for shared values. A value is shared if it is accessible from more than one variable. This means that side effects that change shared values are rare. This gives a functional flavor to large parts of R.

R is a surprisingly rich language with a rather intricate semantics, we can't do it justice in the space at hand. The above summary should suffice for the remainder of this paper.

2.2 The Ecosystem

The largest repository of R code is the Comprehensive R Achive Network (CRAN).[2] With over 12,000 packages, CRAN is a rapidly growing repository of statistical software[3]. Unlike sites like GitHub, CRAN is a curated repository. Each program deposited in the archive must come with documentation and abide by a number of well-formedness rules that are automatically checked at each submission. Most relevant for our purpose, packages must have documentation that comes in the examples and vignettes, and, optionally, tests. All executable artifacts are run at each commit and for each new release of the language. Vignettes can be written using a variety of document processors. For instance Figure 1 contains a (simplified) vignette for a package named A3. That particular vignette combine text formatting comments, bits of documentation with code. For our purposes, the key part is the section tagged `examples`. That section contains code which is a runnable example intended to showcase usage of the A3 package.

While it is remarkable that every package comes with data and a least one vignette, this nevertheless does not necessarily make for good tests. There are two issues with R's approach, vignettes are coarse grained, they typically only exercise top-level functions in a package, and, they do not specify their expected output, sometimes that output is graphics, often it has some textual output. As shown above, vignettes are long-form guides to packages: they describe the problem that the package is designed to solve, and then show how to solve it. Their primary audience is developers. A vignette should divide functions into useful categories, and demonstrate how to coordinate multiple functions to solve problems. Code can be extracted from a vignette and run automatically. One can therefore only assert whether code extracted from examples or vignettes ran without throwing any exceptions and whether the output is similar to the last time this was run, but nothing can be said about the correctness of its output and if there is a difference in output

[1] https://github.com/fikovnik/ISSTA18-Artifact

[2] http://cran.r-project.org

[3] CRAN is receiving about 6 new packages a day [8]

```
% Generated by roxygen2

\name{a3}\alias{a3}

\title{A3 Results for Arbitrary Model}

\usage{a3(formula, data, modelfn, model.args = list(),...)}

\arguments{
 \item{formula}{the regression formula}
 \item{data}{a data frame containing data for model fit}
 \item{modelfn}{the function that builds the model}
}

\description{Calculates A3 for an arbitrary model.}

\examples{
 summary(lm(rating ~ ., attitude))
 a3(rating ~ ., attitude, lm, p.acc = 0.1)
 require(randomForest)
 a3(rating ~ .+0, attitude, randomForest, p.acc = 0.1)
}
```

Figure 1: Sample vignette.

whether this is a substantial departure or if it is an accident of the way numbers are printed.

In our experience, most R developers are domain experts, rather than trained software engineers; this is a possible explanation for the relatively low proportion of unit tests in packages hosted in CRAN. Most tests are simply R scripts that are coupled with a text file capturing expected textual output from the interpreter running the given test [12]. The most popular unit testing framework, called `testthat`, is used by only 3,017 packages (about 23% of the total number of packages in CRAN).[4] In our experience, even when `testthat` is used, only a few tests are defined, covering only a subset of the interface of the package.

2.3 Automating Test Extraction

Our goal is to extract black-box unit tests for packages from a large body of client code. The benefit we are aiming for is to reduce the manual effort involved in the construction of such regression testing suites and not necessarily to find new bugs. To the best of our knowledge there are no tools that address this issue for R, and the previous work requires substantial adaptation to translate to our context.

Test generation is a topic with an extensive body of work. We will only touch on the research directly relevant to our development. We focus on automated techniques that do not require users to write specifications or to provide additional information. The literature can be split into work based on static techniques, dynamic techniques and a combination of both.

Static techniques use the program text as a starting point for driving test generation [1, 5, 16]. The difficulty we face with R is its extreme dynamism. Consider the following expression `f(x/(y+1))`. The semantics of R entails that definitions of `f`, `/`, `+`, and even `(` have to be looked up in the current environment. Given that `x` and `y` may have class attributes, any of those functions could have to dispatch on those. At any time, any of these symbols can be redefined (and in practice they are, sometimes for good reasons [9]). It is also possible for code to reflectively inject or remove variable bindings in any environment and turn any expression back into text that can be manipulated programmatically and then turned back into executable code.

One approach that would be worth exploring, given that sound static analysis for R appears to be a non-starter, is some combination of static analysis and machine learning. For instance, MSeqGen uses data obtained by mining large code bases to drive test generation [19]. So far we have not gone down that road.

Dynamic approaches for generating unit tests often rely on some form of record and replay. Record and replay has been used to generate reproducible benchmarks from real-world workloads [13], and, for example, capture objects that can be used as inputs of tests [6]. Joshi and Orso describe the issues of capturing state for Java programs [7]. Test carving [4] and factoring [15] have very similar goals to ours, namely to extract focused unit tests from larger system tests. These works mostly focus on capturing and mocking up a sufficiently large part of an object graph so that a test can be replayed. Rooney used similar approach to extract tests during an actual use of an application through instrumentation [14]. While R has objects, their use is somewhat limited, they are typically self=contained, and capturing them in their entirety has worked well enough for now.

In terms of pushing our work towards bug finding, we pine for the sanity of languages such as Java or even JavaScript, as in these languages it is not too hard to get an errant program to throw an exception—null pointers or out of bound errors are reliable oracles that something went wrong [3]. In R, most data type mismatches cause silent conversions, out of bound accesses merely extend the target array, and missing values are generated when nothing else fits; none of this causes the computation to stop or is an obvious sign that something went wrong.

Finally, there are few tools dealing with languages that support call-by-need. The most popular tool in that space is QuickCheck [2] for Haskell. It leverage types as well as user-defined annotations to drive random test generation.

3 GENTHAT: DESIGN & IMPLEMENTATION

An *execution trace* is the sequence of operations performed by a program for a given set of input values. We propose to record sets of execution traces of clients of a target package and from those traces extract unit tests for functions in either the public interface or the internal implementation of that target package. To create those unit tests, only data needed to execute individual function calls is required. This boils down to capturing function arguments and any global state the function may access or rely on. To validate that the function ran successfully, outputs must be recorded so that the value observed in the trace can be compared with results of running the generated test.

Consider Figure 2 which is illustrates generation of a test for function `subset` in some target package. It features code from a client of that target package (`a`), the target package (`b`) that is

[4] https://cran.r-project.org/web/packages/testthat

```
> a <- c("Mazda", "Fiat", "Honda")
> b <- c(110, 66, 55)
> cars <- data.frame(name=a, hp=b)


> subset(cars, cars$hp >= 60)

      name  hp
1    Mazda 110
2     Fiat  66
```

(a) Client code

```
subset <- function(xs, p) {
  # filter the data frame rows
  xs[p, ]
}
```

(b) Package

```
test_that("subset", {
  cars <- data.frame(
    name=c("Mazda", "Fiat", "Honda"), hp=c(110, 66, 55)
  )

  expect_equal(
    subset(xs=cars, p=cars$hp >= 60),
    data.frame(name=c("Mazda", "Fiat"), hp=c(110, 66))
  )
})
```

(c) Extracted test

Figure 2: Extracting a test case for `subset`.

```
> a <- c("Mazda", "Fiat", "Honda")
> b <- c(110, 66, 55)
> cars <- data.frame(name=a, hp=b)

# hp is only defined in the cars data frame
> subset(cars, hp >= 60)

      name  hp
1    Mazda 110
2     Fiat  66
```

(d) Client code

```
subset <- function(xs, p) {
  # capture the expression without forcing promise
  expr <- substitute(p)
  with(xs, xs[eval(expr), ])
}
```

(e) Package

```
test_that("subset", {
  cars <- data.frame(
    name=c("Mazda", "Fiat", "Honda"), hp=c(110, 66, 55)
  )

  expect_equal(
    subset(xs=cars, p=hp >= 60),
    data.frame(name=c("Mazda", "Fiat"), hp=c(110, 66))
  )
})
```

(f) Extracted test

Figure 3: Tracing lazy-evaluation and promises.

to be tested, and the corresponding generated test (`c`). The client code creates a data frame with information about vehicles and then filters out cars with horse power less than 60. The `subset` function subsets the given data frame (the `cars$hp >= 60` returns a vector of booleans `c(TRUE,TRUE,FALSE)` that is used to filter out data frame rows in `xs[p, ]`). The generated test first recreates the argument of the call, then it calls the `subset` function and checks the return value.

In a simpler language, all one would need to do would be to record argument and return values as shown in the example of Figure 2. Not so in R. Lazy evaluation complicates matters as argument are passed by *promise*, a promise is a closure that is evaluated at most once to yield a result. Thus, at function call one cannot record values as they are not yet evaluated. The example in Figure 3 shows an improved version of the `subset` function which allows programmers to reference the columns directly so the sub-setting predicate in the client code (`d`) can be more directly written as `hp>=60` instead of `cars$hp>=60`. But, in (`d`) the is no definition for `hp`, so the code would be incorrect if the argument was evaluated eagerly. Instead, in (`e`) reflection is used to reinterpret the argument. The expression `substitute(p)` does not force a promise, instead it simply gets the source code of the expression passed as argument `p`. This is subsequently used in the `with` function that evaluates its arguments in an environment that has been enriched by content of the first argument. Thus `hp>=60` is evaluated in the context of `cars` which has a column named `hp`. Now, the problem for our tool is that we cannot simply record the values of the passed arguments because doing so would force promises that cannot be evaluated in the current frame. In our example, trying to record `p` would force the evaluation of `hp>=60` which would fail since there is not variable binding called `hp` at the scope of the call. In general, manually forcing a promise is dangerous. It may alter the behavior of the program since promises can have side effects. Moreover the reflective capabilities of R dictate that the generated expression must be as close to original call as possible. Any syntactic change may be observable by user code. The generated test thus attempts to retain the structure of the client source code as much as possible.

3.1 GENTHAT Overview

Figure 4 presents an overview of the main phases of GENTHAT:

(a) **Install**: given a target package, the package is downloaded from CRAN and installed in the current R environment. Furthermore, any package in CRAN that transitively depends on the target package is also acquired.
(b) **Extract**: executable code is extracted from the examples and vignettes in the installed packages. Target package functions that are to be tested are instrumented. Depending on which option is selected, either the public API functions or the private ones are decorated. All executable artifacts are turned into scripts. Each script is a self-contained runnable file.

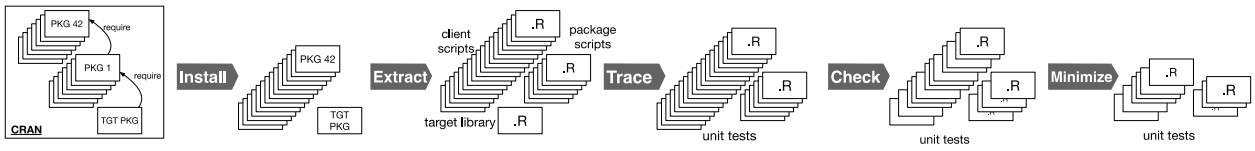


Figure 4: Overview. GENTHAT **installs packages from CRAN, it extracts scripts, traces their execution, and generates unit tests. Unit tests are checked for correctness and validity, and finally tests are minimized.**

(c) **Trace**: for each script, run that script and record trace information. From these traces generate unit tests for all calls to target functions.
(d) **Check**: all unit tests are checked for validity and correctness. Valid tests are those that execute without error. Correct ones are those that return the expected result. Invalid and incorrect tests are discarded. Code coverage data is recorded for the tests that are retained.
(e) **Minimize**: optionally, minimize the test suite. Minimization uses simple heuristics to discard tests that do not increase code coverage. Coverage being equal, tests that are, textually, smaller are preferred.

Test extraction can fail. Section 4.4 details the reasons, but at a high-level, the failures can occur during (a) tracing because the instrumentation perturbs the behavior of the program, (b) generation because some value could not be serialized, (c) validation because deserialization fails, and (d) correctness checking because the test was non-deterministic or relied on external state that was not properly captured. These failures account for the difference in coverage between the original traces and the extracted tests.

We will now describe salient points of our implementation. R is an interpreted language. A straightforward solution would therefore be to modify the R interpreter. However, one of our design goals was to make GENTHAT available as a package in the CRAN repository so that developers can use it without having to modify their distribution. This limits the implementation of GENTHAT to publicly available APIs of the R environment. The implementation is mostly written in R. The only exception is the serializer which is written in C++ for performance reason; a serializer written in R turned out to be an order of magnitude slower.

The whole system consists of 4,500 lines of code and 760 lines of C++ code. The software is released in open source[5].

3.2 Tracing

Tracing involves capturing function invocations and recording the inputs and outputs, including any additional details need to reproduce the function call in an isolation. Consider the following example of a function that filters a vector `xs` by a predicate `p`:

```
filter <- function(xs, p) xs[sapply(xs, p)]
```

Tracing is realized by instrumentation of R functions. Code is injected in target function bodies to record argument values, return value, and the state of the random number generator.[6] After instrumentation, the above function body is rewritten to:

```
`_captured_seed` <- get(".Random.seed", env=globalenv())
on.exit({
  if (tracing_enabled()) {
    disable_tracing()
    retv <- returnValue(default = `_deflt_retv`)
    if (normal_ret(retv)) {
      record_trace(name = "filter", pkg = NULL,
                   args = as.list(match.call())[-1],
                   retv = retv,
                   seed = `_captured_seed`,
                   env = parent.frame())
    }
    enable_tracing()
}})
xs[sapply(xs, p)] # original function body
```

The inserted code has a prologue that runs before any of the original function's code and an epilogue that runs right before the function exits. The latter relies on the `on.exit` hook provided by R. The prologue records the random seed. Due to R's lazy nature, recording of the argument values is done at the end of the function to avoid forcing a promise before it is needed—that might cause a side effect and change the result of the program or cause an error. With the exception of environments, R values are immutable with a copy-on-write semantics, so any modification of function arguments in the function's body will trigger their duplication. There are two things that need to be done before recording a call. First, tracing must be temporally disabled as it could recursive. Next, we need to check whether the function terminated normally or threw an exception. A call is only recorded in the case the function did not fail. While most values can be easily recorded, three type of values must be handled specially: symbols, expressions, and closures. Consider this example:

```
m <- 0.5; n <- 1;
filter(runif(10) + m, function(x) x > n)
```

The code has a call to `filter` with a nested, anonymous function definition which captures the variable `n` from the environment at the call site. Captured variables need to be resolved to their actual values. R has a built-in code analysis toolkit[7] that can report the list of free variables used by a closure. In the above call the set of free variables is `` `{` ``, `runif`, `` `+` ``, `m`, `n`, `` `>` ``. They are resolved to their values using the hierarchy of environments starting with the enclosing environment of the traced function. Extra care is needed for symbols coming from other packages. For those, we do not store values but instead a call to get their values at runtime. In the example, `runif` is a function from the stats package so the

[5] https://github.com/PRL-PRG/genthat
[6] In statistics, random number generators are omnipresent.
[7] https://cran.r-project.org/web/packages/codetools/

call `stats::runif` will be kept. Since it is possible to redefine any symbols including `` `+` `` and `` `>` ``, we have to resolve them as well. However, from the base library, we only keep the symbols that were modified. The complete trace for the above call will therefore be:

```
$ :List of 6
 ..$ fun   : chr "filter"
 ..$ pkg   : NULL
 ..$ args  :List of 2
 .. ..$ xs : language runif(10) + m
 .. ..$ fun: language function(x) x > n
 ..$ globals:List of 3
 .. ..$ runif: language stats::runif
 .. ..$ m   : num 0.5
 .. ..$ n   : num 1
 ..$ seed  : int [1:626] 403 20 -1577024373 1699409082 ...
 ..$ retv  : num [1:3] 0.5374310 1.4735399 0.9317512
```

The very same has to be done for global variables that are closures. For example, if the filter function is called as follows:

```
gt <- function(x) x > n
filter(runif(10) + m, gt)
```

The tracer needs to capture the free variable `n` and store it in the environment of the `gt` function. The `args` and `globals` of this call become:

```
 ..$ args  :List of 2
 .. ..$ xs : language runif(10) + m
 .. ..$ fun: symbol gt
 ..$ globals:List of 3
 .. ..$ runif: language stats::runif
 .. ..$ m   : num 0.5
 .. ..$ gt  : language function(x) x > n
```

The value of `n` is stored in the enclosing environment of the `gt` function.

3.3 Generating Tests

To generate unit tests out of the recorded trace we need to write out arbitrary R values into a source code file and to fill a template representing a unit test with the generated snippets of code. We first describe the general mechanism for serializing values to source code also known as *deparsing*.[8] The `deparse` function turns some values into a character string. Unfortunately, it handles only a subset of the 25 different R data types [10]. Since any of those data types can show up as an argument to a function call, we had to provide our own deparsing mechanism to support them all. In general we strive to output textual forms of arguments because they can be inspected and modified by developers. But there are values for which it is either impractical (*e.g.* large vectors or matrices) or not possible (*cf.* below) to turn them back into source code, for those we fallback to built-in binary serialization. The data types can be divided into the following groups:

Basic values There are 6 basic vector types (logical, integer, numeric, character, complex, and raw). Together with types representing internal functions (*e.g.*, `if`, `+`) as well as the type representing the `NULL` value, they can be turned into source code using `deparse`. The subtleties of floating-point numbers and Unicode characters are therefore handled directly by R.

Lists and symbols Lists are polymorphic, they can contain any value and therefore we cannot use `deparse`. For symbols, `deparse` does not handle empty symbol names and does not properly quote non-syntactic names.

Environments Environments are mutable hash maps with links to their parent environments. They are not supported by `deparse`. Serialization is similar to lists; however, we need to keep track of possible cycles. It is also important to properly handle named environments and exclude the ones that were imported from packages.

Language objects A language object contains the name of the function and a list of its arguments values. Additionally to ordinary functions, there are also unary functions, infix functions, `()`, `[]`, `[[]]` and `{}` all of which needed to be properly handled.

Closures Closures are triples that contain a list of formal parameters, code, and a closing environment. To serialize them properly, we need to resolve all their dependencies which is a transitive closure over its global symbols. We used the same mechanism for recording closures.

S4 S4 is one of the builtin R object systems. S4 classes and objects are internally represented by a special data type. They are complex and are currently serialized in a binary form.

Promises If a promise is encountered, the serialization tries to force it while hoping for the best. They are infrequent, because, when we serialize all the arguments to the function which have been used have been forced.

External pointers External pointers are used for system objects such as file handles or in packages that expose native objects. They are linked to the internal state of the interpreter and there is no general mechanism for persisting them across R sessions. They are currently not supported.

R internal values These values are used within R's implementation. There is no reason these should be exposed in a unit test and thus they are not supported.

Some values might be large in their serialized form. For example, the only way to capture the random number generator is to write out its entire state, which is a vector of over 600 floating-point numbers. If we were to add that to the generated test, it would create ungainly clutter; 126 lines of the test would be full of floating point values that are of no obvious advantage for the reader. Such values are therefore kept separately in an environment that is saved in a binary form next to the test file.

With all the trace values serialized, generating a unit test is merely filling a string template and reformatting the resulting code for good readability. For the latter we use the formatR package[9]. The extracted unit test from the second example call `filter(runif(10) + m, gt)` is listed in Figure 5. The `.ext.seed` referencing an external value which will be loaded into the test running environment at runtime.

[8] In the R world, serialization refers to a binary serialization while deparsing denotes turning values into their source code form.

[9] https://cran.r-project.org/web/packages/formatR/index.html

```
library(testthat)

.Random.seed <<- .ext.seed

test_that("filter", {
  m <- 0.5
  gt <- genthat::with_env(function(x) x > y, list(y=5))
  expect_equal(
    filter(xs=stats::runif(10) + m, fun=gt),
    c(0.5374310 1.4735399 0.9317512)
  )
})
```

Figure 5: Extracted test for `filter(runif(10)+m,gt)`.

3.4 Minimization

Once tests have been created, GENTHAT checks that they pass and discards the failing ones. At the same time we record their code coverage. Given the code coverage we can minimize the extracted test suite, eliminating redundant test cases [21]. Our minimization strategy is a very simple heuristic that aims to decrease the number of tests and to keep the size of individual tests small. In future work we will investigate alternative goal functions such as minimizing the combination of test size, test number, and execution time.

The current heuristic works as follows. Tests are sorted by increasing size (in bytes), with the intuition that, all things being equal, we prefer to retain smaller tests over the larger ones. We use the covr package to compute code coverage during minimization. The approach is to take each test, compute the coverage and compare that coverage to the union of all the previously retained tests. If the new test's coverage is a subset of the coverage so far, the test can be discarded. If the test increases coverage, it is retained. In practice, only a small fraction of tests are retained.

It is technically possible to compute the code coverage during tracing, discarding redundant traces. However, running R code with code coverage is slow and since there are many duplicate calls (only a third of the calls in our experiment were unique) this would impose a significant overhead. Instead we discard duplicate calls during tracing and then run the minimization at the end.

4 EVALUATION

To evaluate GENTHAT, we were interested in finding out how much we can improve test coverage by extracting tests from documentation and reverse dependencies and how efficient such an extraction can be. We were also interested in finding what proportions of the functions calls can be turned into test cases and how large the resulting test suites would become.

For these experiments, we selected 1,700 packages from CRAN. We picked the 100 most downloaded packages from RStudio CRAN mirror[10] including their dependencies and then 1000 random selected CRAN packages, again including their dependencies. This added up to some 1,726 packages. The motivation for this choice is to have some well established and popular packages along with a representative sample of CRAN. From the 1,726, some 1,545 ran succesfully. The remaining 181 failed either because of a timeout (5 hours during tracing), a runtime error, or failure to compute code coverage. The packages amounted to 1.7M lines of R (stripped of comments and empty lines). There were 158,208 lines of examples (on average 102 lines per packages), 32,524 lines of code extracted from vignettes (on average 21 lines per package) and 163,835 lines of tests (on average 106 lines per package).

The experiments were carried out on two virtual nodes (each 60GB of RAM, 16 CPU at 2.2GHz and 1TB virtual drive) using GNU parallel [17] for parallel execution. We used GNU R version 3.4.3[11] from Debian distribution.

4.1 Scale

Table 1 presents an overview of the scale of the experiment in terms of number of function calls and number of extracted tests. Out of 5.3M of function calls, GENTHAT traced 1.6M unique calls, calls with distinct arguments and return values. Out of the recorded traces, 93.6% were saved as unit tests. On average, 86.2% of the generated tests were valid and correct. From the total of 1.3M correct tests 97.9% were redundant, *i.e.* not increasing code coverage. Finally, some 26,838 tests were retained.

Table 1: Function calls & extracted tests for 1,545 packages (*s*=standard deviation, *m*=median). Reproducibility is the ratio between the number of valid and correct tests and the number of traced unique calls.

	Overall	Average per package	
Total number of calls	5,274,108	3,413	(s=13,375 m=141)
Traced unique calls	1,615,151	1,045	(s=3,145 m=82)
Extracted tests	1,512,555	979	(s=3,067 m=68)
Valid & Correct tests	1,303,114	843	(s=2,823 m=50)
Non-redundant tests	26,838	17.4	(s=33 m=9)
Reproducibility	0.8	0.75	(s=0.3 m=0.9)

4.2 Coverage

Figure 6 compares the code coverage obtained with the default tests and the coverage obtained using GENTHAT using code from examples, vignettes and tests. The default tests provide very little coverage for many packages. GENTHAT is able to increase the coverage from an average of 19% to 53% per package. This is a significant improvement.

4.3 Performance

Table 2 presents the average time of tracing and generating tests for our 1,545 packages (tracing), running the generated tests (testing w. GENTHAT), and running all of the code that comes with the package (default tests, vignettes and examples); *s* is standard deviation and *m* is median. In our testbed, running 16 packages in parallel on each node, the tracing took 1d 18h, running genthat tests 19m and running the package code 1h 09m.

Tracing clearly has a significant overhead. The median time is 60 seconds. Given that test generation is a design time activity, this is likely acceptable. The main portion of the time is spent in running

[10] Used data for June 2017 from http://cran-logs.rstudio.com.

[11] Released in November 2017

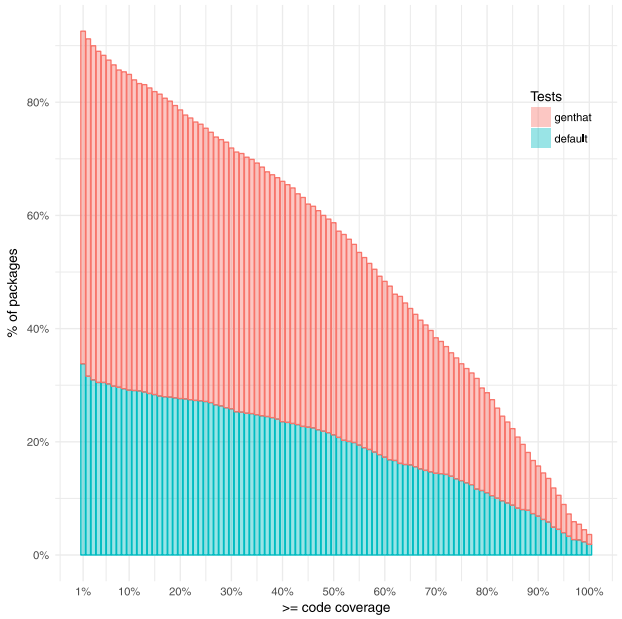


Figure 6: Code coverage from package tests and GENTHAT extracted unit tests. Each bar shows the ratio of packages that have at least that code coverage.

Table 2: Average running time for 1,545 packages.

Task	Average per package
Tracing	420s (s=1500, m=60)
Testing w. GENTHAT	15s (s=29, m=8)
Testing (default)	28s (s=60, m=10)

the generated tests which are numerous. On average, each package yields 1000 tests. They all have to be run to determine validity and correctness, as well as test minimization to determine whether they increase code coverage. On the other hand, testing with the minimized tests runs quickly. We are able to process all the packages in 19 minutes. Thanks to test minimization, this is faster than the time it would take to run all of the original examples, tests and vignettes. Concretely, running GENTHAT extracted tests for the 1,545 is 1.9 times faster. The resulting coverage is 15% smaller, but what we provide are actual unit tests while the code in examples and vignettes (and often in R tests as well) is just a plain R code with no assertion of expected behavior other than that it runs.

4.4 Accuracy

In terms of accuracy, GENTHAT managed to recreate 80% of all the unique function calls. Here we discuss the remaining 20%. Problems can be divided into three groups: tracing, generation, and replay. They are summarized in Table 3.

The tracing inaccuracies happen while recording function arguments. They account for 4.3% of the failures. Most are due to trace size; we intentionally discard values larger than 512KB in the R

Table 3: Issues in 312,037 failed test extraction.

	Overall	%
Tracing	70,020	4.3%
- skipped traces (size > 512kB)	60,360	
- . . . used in an incorrect context	2,150	
- unable to resolve symbol	1,419	
- unclassified	6,091	
Test generation	32,576	2.0%
- environments with cycles	24,200	
- other serialization errors	1,655	
- unclassified	6,721	
Test replay	209,441	12.9%
- incorrect tests (value mismatch)	77,850	
- invalid tests (execute with error)	131,591	

binary data format. Our reasoning is that tests which are too large are awkward to work with. The size limit was chosen empirically such that no single package loses more than 10% of its tests. The largest test was 887 MB (on average, skipped tests were 3.5 MB). Among the other tracing inaccuracies are the triple-dots (. . .) used in an incorrect context and missing symbols. Both problems are related to non-standard evaluation [20] in which the function arguments are not meant to be resolved in the usual scope. In most cases, GENTHAT records arguments correctly, but sometimes (0.2%) it fails to resolve them properly. The remaining problems are difficult to classify; many issues are related to some of the corner cases and issues with GENTHAT itself. They contribute very little to the overall failure rate.

The test generation inaccuracies are related to environment serialization in the presence of cycles. This happens for example if there is a variable in an environment that references the same environment or any of its parents. This is a known issue that shall be fixed in future version of GENTHAT. Other serialization problems are related to unsupported values such as weak references or byte code. Similarly, to the unclassified tracing errors, the rest of the problems does not have a common cause. Some problems come from the `deparse` function. Again, they occur very rarely.

The final category is responsible for most of the inaccuracies. They are the most difficult to reason about since they require knowledge of the actual package code. However, there are some common causes:

- External pointers (pointers to native code) are not supported, so any functions that use them explicitly or transitively will not work.
- We do not keep any state, so if there is an expected sequencing for function calls GENTHAT will not work (for example, `close` must follow `open`).
- Non-standard evaluation and serialization do not always play well together. Currently, GENTHAT does not have any heuristics to guess in which evaluation mode a function uses and thus the serialization might fail.

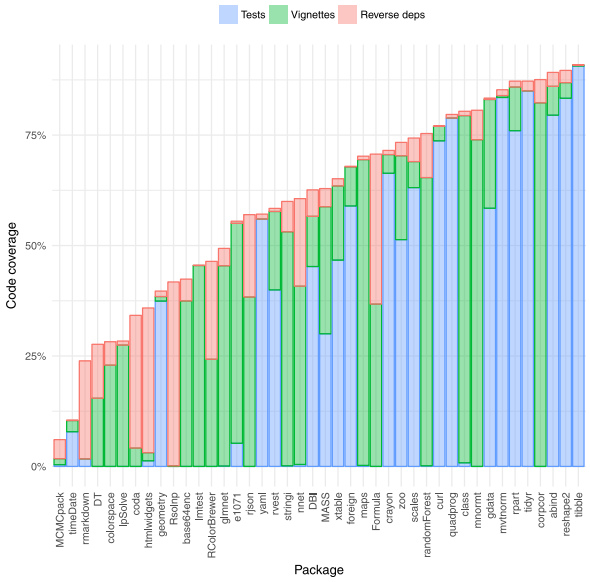


Figure 7: Code coverage obtained with reverse dependencies.

– Out of the 4 popular object systems used in R, we only cover the builtin S3 and S4. Reference classes and R6[12] are currently not supported.

There is one last source of inaccuracy. The function calls tracing happens in the `on.exit` hook that is injected into method bodies. While this mostly works, the hook can be overwritten and the entire call will be missed. To quantify the frequency of such event, we instrumented the code with a counter to count the number of function calls. In the 1,545 packages, the counter registered 5,274,108 function calls, on average 3,413 per package. The `on.exit` hook registered 245,815 fewer calls (5% less). Currently, the `on.exit` hook is the only way to inject code into a R function without changing the R interpreter. We have raised this question of how to avoid hooks being overridden with the Core R team and a solution will be hopefully included in an upcoming release.

4.5 Reverse Dependencies

One of the ideas we wanted to explore is to use the code found in clients of a package to generate additional unit tests. As computing test for the transitive closure of a package's reverse dependencies can be quite expensive we randomly selected 65 packages among all packages that had at least 20 reverse dependencies. For each of these package, we downloaded all of the packages that depend on it, extracted scripts from those packages tests, vignettes and examples, and used those scripts to extract test cases. In 42 cases, using reverse dependencies improved code coverage. On average, the coverage increased from 52% to 60%. Figure 7 is a histogram of the cumulative test coverage obtained from only the default tests, the GENTHAT results for the package only, and the GENTHAT results including all reverse dependencies. The improvements are significant in some cases.

[12] https://cran.r-project.org/web/packages/R6/

5 DISCUSSION

Following the evaluation we present the threats to validity of the selected approach, propose some of its application and discuss a possible generalization.

5.1 Threats to Validity

There are certain limitations to our work. Some can be mitigated by more engineering work, some is intrinsic to the approach.

Large tests. Programs in R naturally tend to use large vectors and matrices. Serializing them into source code results in rather a large unreadable chunks of tests which have a little value for the reader. On average the generated tests are 18.58 kB (median 490.00 B). The largest test that was generated was 1.55 MB of R code. GENTHAT can already be tuned to discard traces that exceed certain size already at runtime (*cf.* Section 4.4). Further, large values can be separated into the `.ext` file keeping only a reference in the unit test code.

Non-determinism. If a function call includes non-determinism which is not captured by the value of calling arguments, the generated tests will most likely fail.

Over specific testing oracle. The resulting unit tests directly represents the code from which they were extracted. They are as good as the source code it. If a package does not include much runnable artifacts or does not have many clients, GENTHAT will not do much.

Brittleness. Extracting tests from existing code can in general produce false positives (extracted tests passes while the original code from which it is generated fails) or false negatives (extracted tests fails while the original code runs) [14]. False positives do not occur in GENTHAT since if a function call fails, GENTHAT does not keep the trace (*cf.* Section 3.2). False negatives, apart from the reasons discussed in 4.4, also occur when the tests are run on a different version of a package than the one that was used to extract the test suite. If the code base changes, the extracted tests naturally becomes out of date. The same happens for manually written tests. The advantage of the automatically extracted tests is that the process can be simple repeated. On the other hand, GENTHAT does not currently have any support for simple regenerating tests that are out of date.

Tracing time. The tracing comes with some overhead. On average, tracing is 21.5 (s=97.1 m=4.9) times slower than simply running the package code artifacts. For most packages, the running time reasonably short, 75% of them runs under 4m. But there are packages for which the overhead makes the running very long, with some exceeding the set timeout of 5h. Commonly, this happens in packages that contains some iterative algorithms that run on a sufficiently large matrices. Since most of these calls are from the execution code path very similar, it shall be possible to mitigate this problem by using stochastic tracing. Instead of tracing each an every function call, the tracing should be guarded by some probability with long-tail distribution.

5.2 Utility

There are multiple applications for the extracted tests. First, they help regression testing. Especially in the case when it is possible to extract code from package clients. GENTHAT can also help to bootstrap manual tests by first generating an initial test suite which then package authors updates by hand. This approach can also be useful for Core R developers and package repositories where packages need to be often tested (for example with every change in the R interpreter). Since the extracted tests runs faster than package artifacts it might be beneficial to use them. They might also help to narrow root cause of a failure since they are actual unit tests.

5.3 Generalization

Text extraction from runtime program traces is naturally going to work better in language/programming models that limit mutable global state and where arguments to functions tend to be self contained. We would expect that scientific programming languages like MATLAB or Julia, and functional languages like Haskell or OCaml should behave in similar ways to R. Imperative languages are likely to be less conducive to this approach. Whether our approach would yield reasonable results in C or C++ is an open question.

6 CONCLUSION

We have presented the design, implementation and evaluation of GENTHAT, a tool for the automated extraction of unit tests for the R language. Our evaluation of this tool on a large corpus of packages suggests that it can significantly improve coverage of the code being tested.

In a way our results are surprising, as the limitations of our tool are quite severe. GENTHAT does not keep any global state or external state, so any test that changes values in a lexically scoped environment is bound to fail. Any code that manipulates pointers to C data structures is likely to fail. And lastly, any code that is not deterministic will surely fail. Our intuition is that GENTHAT works well because of the functional nature of R, mutation is rarely used, function access values passed in, and produce new ones.

For future work, we aim to explore techniques geared towards finding new bugs. This can be achieved by fuzzing the inputs to a function. We also will look for more changes that allow to reduce the inaccuracies of GENTHAT as well as to speed up the process of handling reverse dependencies.

ACKNOWLEDGMENT

The authors thank Petr Maj, Roman Tsegelskyi, Filippo Ghibellini, Michal Vácha and Lei Zhao for their work on previous versions of GENTHAT. We also thank Tomáš Kalibera from the R Core Team for his feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412) and the NSF Darwin Award. The experiment presented in this paper was conducted using GNU parallel [18].

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/566171.566191
[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/351240.351266
[3] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (2004). https://doi.org/10.1002/spe.602
[4] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/1181775.1181806
[5] Michael Ernst, Sai Zhang, David Saff, and Yingyi Bu. 2011. Combined Static and Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2001420.2001463
[6] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl Chang. 2010. OCAT: object capture-based automated testing. In *nternational Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/1831708.1831729
[7] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *International Conference on Software Maintenance (ICSM)*. https://doi.org/10.1109/ICSM.2007.4362636
[8] Uwe Ligges. [n. d.]. 20 Years of CRAN (Video on Channel9. In *Keynote at UseR!*
[9] 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
[10] R Core Team. 2017. R Internals 3.4.3. https://doi.org/3-900051-14-3
[11] R Core Team. 2017. R Language 3.4.3. https://doi.org/10.1016/0164-1212(87)90019-7
[12] R Core Team. 2017. Writing R Extensions 3.4.3. https://doi.org/10.2135/cropsci1998.0011183X003800020005x
[13] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2048066.2048119
[14] Thomas Rooney. 2015. *Automated Test Generation through Runtime Execution Trace Analysis.* Master's thesis. Imparial College London.
[15] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for Java. In *International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/1101908.1101927
[16] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/1081706.1081750
[17] O. Tange. 2011. GNU Parallel–The Command-Line Power Tool. *;login: The USENIX Magazine* 42, 47 (2011).
[18] O. Tange. 2018. GNU Parallel 2018. https://doi.org/10.5281/zenodo.1146014
[19] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-oriented Unit-test Generation via Mining Source Code. In *European Software Engineering Conference and Symposium on The Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/1595696.1595725
[20] Hadley Wickham. 2014. *Advanced R* (1 edition ed.). Chapman and Hall/CRC.
[21] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

A.2 Designing Types for R, Empirically

Designing Types for R, Empirically by *Alexi Turcotte, Aviral Goel, Filip Křikava and Jan Vitek* published in *Proceedings of the ACM Programming Languages 4, OOPSLA*, Article 181, November 2020 [Turcotte et al., 2020c].

Designing Types for R, Empirically

ALEXI TURCOTTE, Northeastern University, USA
AVIRAL GOEL, Northeastern University, USA
FILIP KŘIKAVA, Czech Technical University, Czechia
JAN VITEK, Northeastern University, USA and Czech Technical University, Czechia

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 data analysis, but a bad fit for tools such as compilers or program analyzers. In particular, R has no type annotations, and all operations are dynamically checked at runtime. The starting point for our work are the two 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 from the Kaggle data science 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 Křikava, and Jan Vitek. 2020. Designing Types for R, Empirically. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 181 (November 2020), 25 pages. https://doi.org/10.1145/3428249

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 intuition, as many languages are intended for general purpose programming tasks. Unfortunately, intuition may fail us when looking at domain-specific languages designed for a particular group of users to solve 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 statistical routines written in Fortran. Over three decades they became widely used across many fields for exploratory data analysis. Modern R is fascinating as an object of study. It is a vectorized, dynamically typed, lazy functional language with limited side-effects, extensive reflective facilities and retrofitted object-oriented programming support.

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 salient ones, the lack of type annotations, the

Authors' addresses: Alexi Turcotte, Northeastern University, USA; Aviral Goel, Northeastern University, USA; Filip Křikava, Czech Technical University, Czechia; Jan Vitek, Northeastern University, USA, Czech Technical University, Czechia.

2475-1421/2020/11-ART181
https://doi.org/10.1145/3428249

ability to use syntactic shortcuts, and widespread conversion between data types. While these choices have decreased the barrier to entry—many data science educational programs do not teach R itself but simply introduce some of its key libraries—they also allow for errors to go undetected.

Retrofitting a type system to the R language would increase our assurance in the result of data analysis, but this requires facing 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.

The goal of this paper is to gather data that can be used as input to the process of designing a type system for R. To focus our work, we chose to limit the scope of investigation on the design of a language for function type signatures. Thus, this paper will use the collected data to document the signatures of user-defined functions. For this, we design a simple type language that matches the R data types. We then extract call and return types from execution traces of a corpus of widely used libraries, and finally synthesize type signatures. This allows us to see how far one can get with a simple type language and identify limitations of our design. We validate the robustness of the extracted type signatures by implementing a contract checker that weaves types around their respective functions, and use end-user code and clients of the target packages for validation.

To sum up, we make the following contributions:

- We implemented scalable and robust tooling to automatically extract type signatures and instrument R functions with checks based on their declared types.
- We carried out a corpus analysis of 412 widely used and maintained packages to synthesize function type signatures and validated the robustness of the signatures against 160,379 programs that use those functions.
- We report on the appropriateness and usefulness of a simple type language for R.
- Our data and code are open source and publicly available.[1]

2 BACKGROUND

This section introduces related work and gives a short primer on R.

2.1 Related Work

Dynamic programming languages such as Racket, JavaScript, PHP and Lua have been extended post factum with static type systems. In each case, the type system was carefully engineered to match the salient characteristics of its host language and to foster a particular programming style. For example, Typed Racket emphasizes functional programming and migration from untyped to fully typed code [Tobin-Hochstadt and Felleisen 2008], Hack [Verlaguet 2013] and TypeScript [Bierman et al. 2014] focus on object-oriented features of PHP and JavaScript, respectively. They allow users to mix typed and untyped code in a fine-grained manner. In the case of Lua [Maidl et al. 2014], the type system tried to account for the myriad ways Lua programmers use tables. Other languages adopted a mix of typed and untyped code by design. In Julia, type annotations are needed for method dispatch and performance [Bezanson et al. 2018]. In Thorn, users could freely move between typed and untyped code thanks to the addition of like types [Wrigstad et al. 2010]. Lastly C# is an example of a statically typed language which added a dynamic type [Bierman et al. 2010].

But what if the design of the type system is unclear? Andreasen et al. [2016] propose a promising approach called trace typing. With trace typing, a new type system can be prototyped and evaluated

[1] github.com/PRL-PRG/propagatr, github.com/PRL-PRG/contractr, data: doi.org/10.5281/zenodo.4091818, artifact: doi.org/10.5281/zenodo.4037278

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. We take inspiration in the aforementioned works but focus on adapting them to our target language.

2.2 The R Language

The R Project is a key tool for data analysis. At the heart of R is a vectorized, dynamic, lazy, functional, object-oriented language with an unusual combination of features [Morandat et al. 2012]. R was designed by Ihaka and Gentleman [1996] as a successor to S [Becker et al. 1988].

R's main data type is the primitive vector. Vectors are explicitly constructed by the constructor `c(...)`, as in `c(1L, 2L, 3L)` which creates a vector of three integers. R has a builtin notion of type that can be queried with the `typeof` function. Figure 1 lists all of the builtin types 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`, an `integer` will be coerced to `double`, and a `double` will be coerced to `complex`. Some odd conversions 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 are monomorphic, except `list` which can hold values of any type, including `list`. For all monomorphic data types, attempting to store a differently-typed value will cause a conversion: Either the value is converted to the type of the vector, or vice versa.

All vectorized data types have a distinguished missing value denoted by `NA` (for "not available"). The type of `NA` is `logical` (`typeof(NA)=="logical"`), but `NA` inhabits every type: `typeof(c(1,NA)[2]) == "double"`. R also has a `NULL` value. In data science, it is useful to have a notion of a "missing observation", since vectors are monomorphic there is a need for a missing value to inhabit each primitive type. In a sense, `NA` represents a missing data point, whereas `NULL` represents a missing data set.

Over time, programmers have found the need for a richer type structure. R supports this with *attributes*. One may think of attributes as an optional map from names to values attached to any built in type. Attributes are used to encode various type structures, and they are queried with `attributes`. Using attributes, programmers can extend the set of types by tagging data. For example, take the vector of four values, `x<-c(1,2,3,4)` and attach attribute `dim`, `attr(x,"dim")<-c(2,2)`, to treat `x` as a 2x2 matrix.

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

Another attribute that can be set is the `class`, which is one way to use R for object-oriented programming. The class attribute can be queried with `class`, and it can be bound to a list of class names: For instance, `class(x)<-c("human", "friend")` will set the class of `x` to be `human` and `friend`. 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. 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.

R functions have a number of quirks. Arguments can be assigned arbitrary expressions as default values; functions take variable numbers of parameters, and can be called positionally and nominally. Consider this example:

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

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

```
> f(1)
  [1] 2           # a double vector, y is 0, ... is missing
> 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 polymorphism: `f` may return a vector of integers or of doubles of length equal to the max length of its arguments.

3 A TYPE LANGUAGE FOR R

In this section, we set out to propose a candidate design for a type language to describe the signatures of R functions. The goal is not to propose a final design, but rather a starting point for an iterative process.

Given the peculiarities of the language there are a number of design choices that need to be reviewed and evaluated. It is not controversial to include types that cover the six kinds of primitive vectors, furthermore environments and the distinguished null types are commonly used and must be included. 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. For simplicity, we omit some of the data types that closer to the implementation of the language. Fig. 2 presents our type language.

$$
\begin{array}{rcll}
T & ::= & \textbf{any} & \textit{top type} \\
 & | & \textbf{null} & \textit{null type} \\
 & | & \textbf{env} & \textit{environment type} \\
 & | & S & \textit{scalar type} \\
 & | & V & \textit{vector type} \\
 & | & T \mid T & \textit{union type} \\
 & | & ?\,T & \textit{nullable type} \\
 & | & \langle A_1, \ldots\ A_n \rangle \rightarrow T & \textit{function type} \\
 & | & \textbf{list}\langle T \rangle & \textit{list type} \\
 & | & \textbf{class}\langle ID_1, \ldots\ ID_n \rangle & \textit{class type}
\end{array}
\qquad
\begin{array}{rcll}
A & ::= & T & \textit{arguments} \\
 & | & \texttt{...} & \textit{dots} \\
V & ::= & S\texttt{[]} & \textit{vector types} \\
 & | & \texttt{\^{}}S\texttt{[]} & \textit{na vector types} \\
S & ::= & \textbf{int} & \textit{integer} \\
 & | & \textbf{chr} & \textit{character} \\
 & | & \textbf{dbl} & \textit{double} \\
 & | & \textbf{lgl} & \textit{logical} \\
 & | & \textbf{clx} & \textit{complex} \\
 & | & \textbf{raw} & \textit{raw}
\end{array}
$$

Fig. 2. The R type language

Scalar. While R does not have scalar data types, there are cases where functions expect scalar values, for example a conditional takes a single logical and will complain if more values are passed. We considered tracking dimensions of data structures, but decided against it. Instead, the type language differentiates between vectors of lenght 1 and vectors of any dimension. The primitive types can be either vectors (e.g., **int**`[]`) or scalars (e.g., **int**). A vector can happen to be of length 1, which we handle with subtyping, shown later. Vectors are monomorphic, a vector of doubles contains only doubles.

Missing. Each primitive type has its specific `NA`. Many built-in functions, especially those implemented in C or Fortran, do not support `NA` values. It is thus advantageous to distinguish between vectors that can and cannot contain missing values. In our experience, functions that expect scalar values tend to not admit `NA`s, thus scalar types are treated as being `NA`-free. The type language allows one to write `^`**int**`[]` to specify that a vector of integers *may* have `NA`s and **int**`[]` to say that a vector must not have missing values. The type **raw** does not allow `NA`s, so `^`**raw**`[]` is `NA`-free. R does not provide a built-in type testing mechanism, e.g., for the case of `NA`-free data types, it is necessary to scan vectors to find out if they have missing values.

Nullable. The **null** type is inhabited by a singleton `NULL` value often used as a sentinel. Unlike in some other lanugage, `NULL` is not the default value of uninitialized variables. R has different notion for that (which we do not cover here). To capture common uses of `NULL`, the type language has a nullable type, written $?\,T$. Values of this type can be either values of type T or `NULL`.

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 type language allows one to specify that a value is a list containing element of some type T, written **list**$\langle T \rangle$. R does not have built-in type tests for this purpose, to establish the type of a value requires traversal and checking individual elements.

Class. R has three objects systems, code-named S3, S4 and R5. All of them operate by adding a class attribute to values. The most widely used system by far is S3, which supports single dispatch and multiple classes [Morandat et al. 2012]. The challenge from a type system point of view is that a value (such as an integer `5L`) could be attributed with a class. Code that performs dispatch would use the class attribute while code that does not would view the value as an integer. The type language focuses on the attribute, and will hide the underlying type of the value. While this seems to match common usage, it does represent a loss of expressiveness. The type language also focuses on S3, and we leave the other object systems for future work. S3 has no notion of inheritance, each

value has a list of classes. The type language thus allows to write **class**$\langle ID_1, \ldots, ID_n \rangle$ to denote values tagged with the class names ID_1 to ID_n.

Union. We support untagged unions of types written $T_1 \mid \ldots \mid T_n$. The elements of unions are not disjoint, e.g., 1L is both an **int**, a **int**`[]` and a `^` **int**`[]`.

Function. Functions signatures the form $\langle A_1, \ldots, A_n \rangle \rightarrow T$ where each A_i argument is either a type T or dots (`...`), a variable length argument list. Moreover, a single function's signature can be the disjunction of a number of individual signatures.

Overall, the choice of type language follows the structure of values. The presence of `NA`-free data types and scalars are two choices that must be validated in practice. Note that nullable types are really just a special case of unions (i.e., `?` T is shorthand for $T \mid$ `NULL`).

3.1 Subtyping

R does not support the notion of subtyping between values, but we include a few simple rules in our framework to capture widespread coercion. The conversions between primitive types give us a starting point (e.g., an **int** is always accepted where a **dbl** is expected). Furthermore, the types introduced above induce some the rules of Figure 3: an `NA`-free vector is a subtype of its `NA`-ful equivalent, a value of type T is a subtype of a nullable T, a list is subtype of another list if their elements are subtypes, and a scalar is a subtype of a vector of the same primitive type.

$$
\begin{array}{rcll}
S\texttt{[]} & <: & \texttt{\^{}}\,S\texttt{[]} & \\
T & <: & \texttt{?}\ T & \\
\textbf{list}\langle T \rangle & <: & \textbf{list}\langle T' \rangle & \textit{iff}\ T <: T' \\
S & <: & S\texttt{[]} & \\
\textbf{lgl} & <: & \textbf{int} & \\
\textbf{int} & <: & \textbf{dbl} & \\
\textbf{dbl} & <: & \textbf{clx} &
\end{array}
$$

Fig. 3. Subtyping

3.2 Synthesizing Signatures

Later in this paper, we will introduce a tool to synthesize function type signatures from running R code, wherein each invocation of a function will generate a trace representing the types of function arguments and returns. These traces will be combined into an overall arrow type for the function, and in order to keep these signatures compact we compact the traces into a single top-level arrow with unions at each argument position. Thus, the shape of function signatures will be:

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

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 we adopted.

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 replace **null** types with nullables. Higher-order functions are conservatively treated as **any** $\rightarrow$ **any**.

It is noteworthy that by performing this compaction we are losing precision, and the synthesized signatures will suggest combinations of argument types that were not observed.

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

Fig. 4. Simplification rules

4 ANALYSIS AND INSTRUMENTATION PIPELINE

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 5 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.

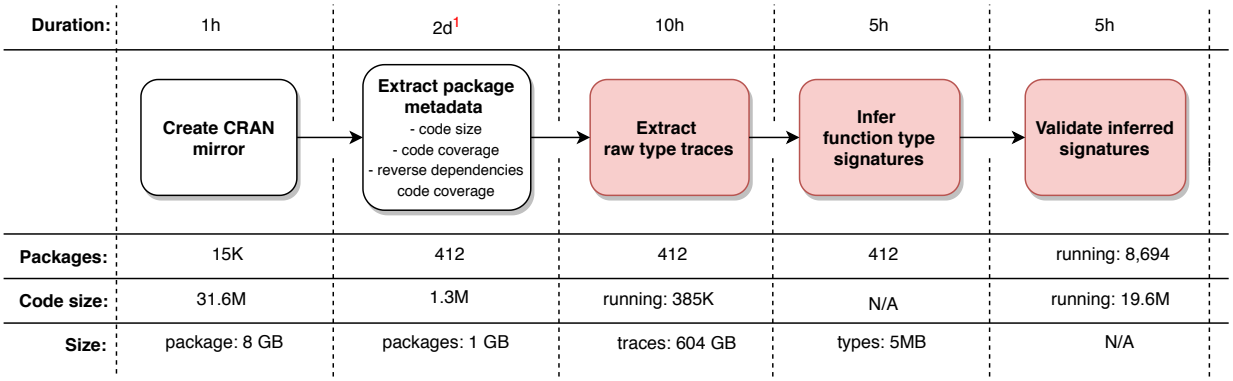


Fig. 5. The Analysis Pipeline; [1] metadata has to be extracted for *all* CRAN packages.

4.1 Types from Traces

We implemented Typetracer, an automated tool for extracting types 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 seemingly simple, the details and their proverbial devil are surprisingly tricky to get right at scale. 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.

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 language. 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: (a) for vectors, the presence of `NA` values, (b) for lists, element types by a recursive traversal, and (c) for promises, an approximation of the

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

Promises. The fact that arguments are lazy (expressions are packed into promises and only evaluated on first access) complicates information gathering. For example, some promises may remain unevaluated, 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 and update the recorded type if the promise is forced.

Missing Arguments. Parameters which receive no values when the function is called are termed missing (not to be confused with `NA`). This occurs when a function is called with too few arguments and no default values are specified for those missing arguments. We record a `missing` type for such arguments. There are two obvious ways to deal with missing arguments: type them as **any** or type them as some unit type. 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.

Dots. Arguments that are part of a dots parameter (denoted `...`) are ignored. We do not attempt to give dots a type.

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 Checking Signatures with Contracts

One can validate a function's type signature by checking that it is respected in all programs that call the function. For this, we developed ContractR, an R package that decorates functions with assertions. We use it to insert type checking code around functions. For speed, ContractR's primary logic is implemented in C++. It has been tested with GNU R-3.5.0 and hardened with a battery of 400 test cases. An invocation of `library(contractr)` causes contracts to be injected. ContractR scans all packages in the workspace and inserts contracts in functions for which type signatures are available. Package load hooks are executed when new packages are loaded. 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 in comments using Roxygen2 annotations, using the `@type` tag:

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

The injected contracts check arguments with a simple tag check when possible. Some properties require traversing data structures, such as the absence of `NA`. For union types, multiple checks may

be needed, at worst one per member of the union. In order to retain the non-strict semantics of R, the expression held in a 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 a callback is added to the exit hook. Another wrinkle is due to 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

Our aim is to gather data with which we can validate our type language design. To that end, we propose an experiment wherein we use Typetracer to synthesize types for some core corpus of packages, and validate those types by using ContractR and installing contracts on core corpus functions and running the client code of this 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 6 shows these packages: the size of the dots reflects the project's size in lines of code including both R and native code[2], 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.

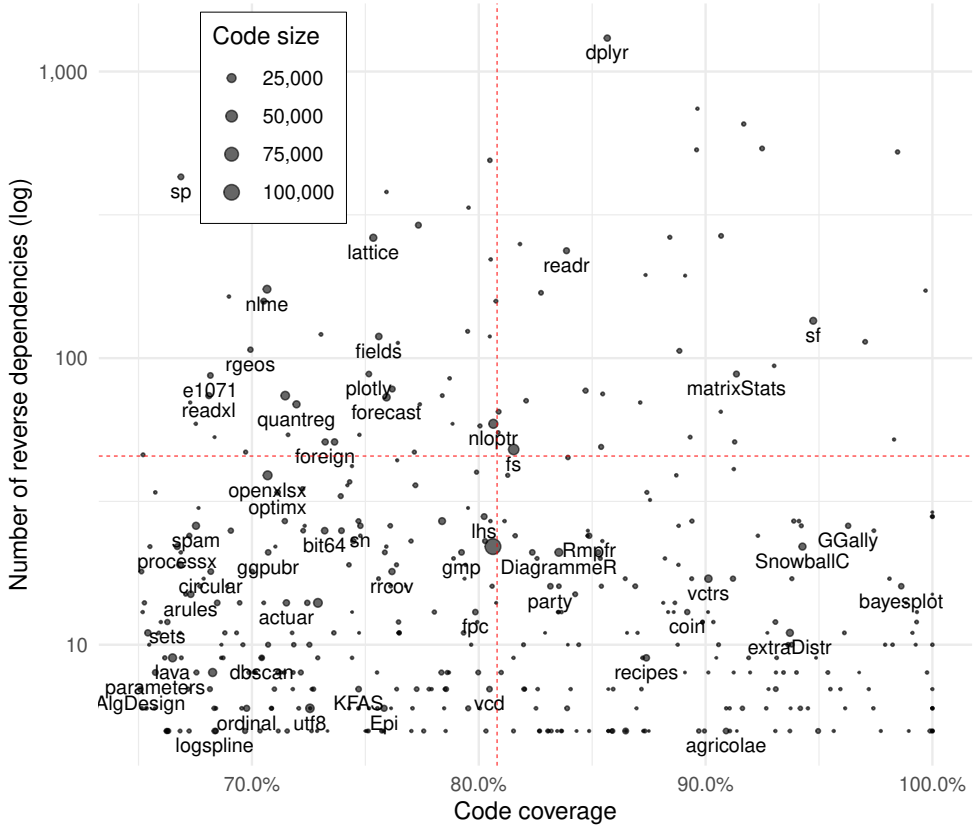


Fig. 6. Corpus

[2]Lines of source code reported excludes comments and blank lines, counted by *cloc*, *cf.* https://github.com/AlDanial/cloc

These packages come from the Comprehensive R Archive Network (CRAN[3]), the largest repository of R code with over 15.4K packages[4] 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.

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 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 allow 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[5], 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[6] 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,[7] 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

[3] http://cran.r-project.org
[4] CRAN receives about 6 new package submissions a day [Ligges [n. d.]]
[5] https://github.com/r-lib/covr
[6] https://www.kaggle.com/c/titanic
[7] We use rmarkdown to convert from notebooks to R.

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.

Table 1. Top types of arguments in 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

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 behavior. 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.

Table 2. Select Type Signatures

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**[]

This section attempts to evaluate how well the proposed type language 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 language is with respect to common programming patterns occurring in widely used R libraries.

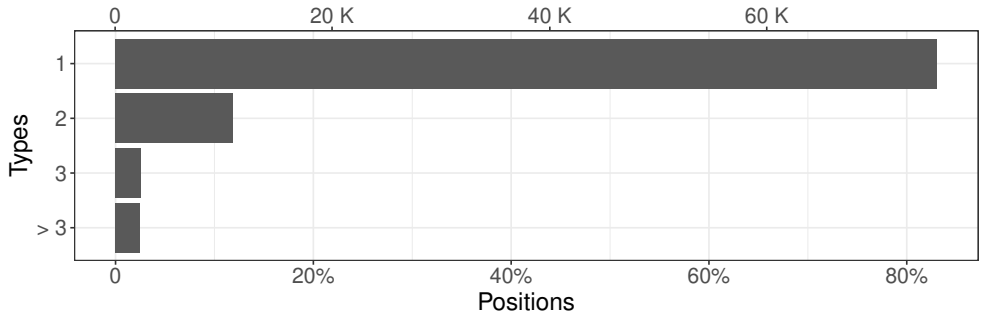


Fig. 7. Size of unions

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's signature. We get to that number in two steps. Fig. 7 shows the number of inferred argument types and their size (in terms of members of the union). The figure shows that most functions do not require a union at all (83.1% of arguments do not have a union), and only 2.5% of positions have unions with more than three members.

Table 3. Singleton Type Categories

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 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 inferred to be **`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.

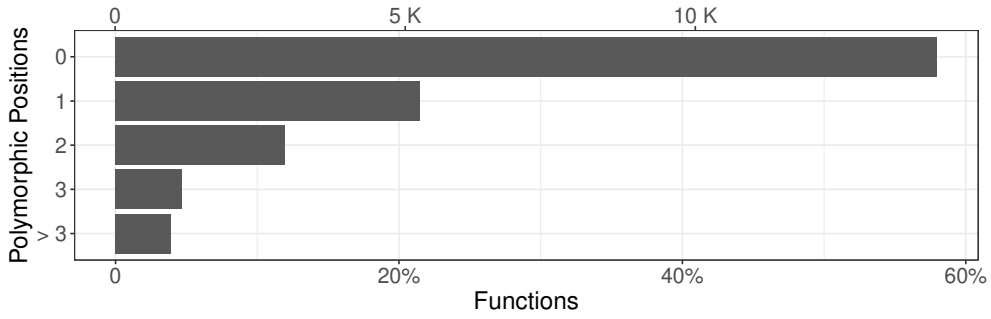


Fig. 8. Function Polymorphism

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 8 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, where the `data(sweetpotato)` expression loads a sample data set to test the function. 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 $\mathrm{dbl}[]$ has type $\mathrm{int}[]$, 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

Table 4. Top contract failures

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

type language 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.

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 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. These are all from 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 for these contract failures: the types that Typetracer generates are only as good as the test code its run on.

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%.

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

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

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

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

Passed	Arg Type	# Arg Types Failed	# Arg Types Total	% 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

Had we failed to capture some key usage pattern of R with our type annotation framework, we would likely see it here. For example, 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 behavior 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 behavior 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 $[1, 2, \ldots, n]$. 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, this evaluation did not reveal any new insights, 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.

Table 8. Top contract failures in Kaggle kernels

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

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

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

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

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**⟨_, 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 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[8] is to use the `stopifnot` function from the R base package. It takes a number of R expressions which should all evaluate to `TRUE`, otherwise a runtime exception is thrown with a message quoting the 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))
```

Beyond `stopifnot`, there are 4 packages in CRAN[9] 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 *assertthat* is maintained and used (with 211 reverse dependencies). The advantage of *assertthat* over the R default is that it provides much better error messages.

One way to assess the usefulness of our type checking framework is to see how many existing user-defined type checks can be replaced by ContractR. To measure this, we extracted all calls to `stopifnot` and `assertthat` assertions, and checked which among them could either be completely replaced by ContractR, or at least partially simplified by removing a portion of an assertion expression. Partial simplification is useful, as a common pattern in this ad hoc type checking is that the first part of the assertion checks the value's type, while the rest checks more detailed properties. In the example above, the whole expression could be replaced by a **`chr`** type check.

Out of the 412 packages, 153 use runtime assertions. Altogether, there are 1,995 assertions in 1,264 functions. Among these, ContractR can replace 1,005 (50.4%) assertion calls across 114 packages and 688 functions. Furthermore, an additional 1,223 (61.3%) asserts in 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 defined 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 DISCUSSION

Throughout this paper, we employed the following strategy to consolidate types: We collected all of the traces observed for a function, and combined them into a single function type, where the type for each argument position was made up of a union of all the observed types at that position. We then simplified these unions using the rules presented in Section 4. We call this an *arrow of unions*. We will elaborate on some of the design decisions that went into developing this method.

Arguably, the primary issue with the arrow of unions strategy is that of a loss of precision, as relationships between argument and return types can be obscured. For example, consider a function with two unique call traces, ⟨**`chr`**⟩ → **`chr`** and ⟨**`dbl`**⟩ → **`dbl`**. The arrow of union strategy results in a combined type of ⟨**`chr`** | **`dbl`**⟩ → **`chr`** | **`dbl`** for the function, which hides the (potential) relationship between the argument and return type.

One solution to this is simply to convert each call trace into an arrow type, and generate a union of arrow types as the overall type for the function, dubbed the basic *union of arrows* strategy. This would lead to the type ⟨**`chr`**⟩ → **`chr`** | ⟨**`dbl`**⟩ → **`dbl`** for the aforementioned function. Unfortunately, this leads to a significant blow-up in the size of types, and makes many types unreadable due to the

[8] 87.6% of all runtime checks in the whole of CRAN

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

myriad ways in which we combine the primitive types, as recall that, e.g., scalars and vectors are not the same type, and we see many types of the form ⟨**chr**⟩ → **dbl** | ⟨**chr**[]⟩ → **dbl** | ⟨ˆ**chr**[]⟩ → **dbl**. When comparing this consolidation strategy against the one employed throughout the paper, we find that the types using this strategy are much larger, with 14,057 (55.06%) functions having at least one top-level union. We observed 93,229 independent arrow types using this strategy.

Another option is to employ a hybrid approach, wherein we perform the union of arrow types *after* grouping them together by return type (to further simplify, we also combine some primitive return types together, such as **dbl** and **dbl**[]). While this has the advantage of reducing most of the blow-up of the previous approach, it suffers particularly when functions return classes, as our type system does not allow us to effectively consolidate class types. This strategy would reduce arrow types ⟨**chr**⟩ → **dbl**, ⟨**chr**[]⟩ → **dbl**, and ⟨ˆ**chr**[]⟩ → **dbl** into the function type ⟨ˆ**chr**[]⟩ → **dbl**. Comparing again to the strategy employed throughout the paper, we find 5,317 (20.82%) functions to have a union of arrows, and we found 38,650 independent arrow types. We additionally find that 38,650 (26.14%) of arrow types have at least one polymorphic argument.

These findings are summarized in Figure 9, which shows a breakdown of the number of *top-level alternatives* in the function types obtained with both of these strategies: The term "top-level alternative" signifies an arrow type in the union of arrows, e.g., ⟨**chr**⟩ → **chr** and ⟨**dbl**⟩ → **dbl** are top-level alternatives in the type ⟨**chr**⟩ → **chr** | ⟨**dbl**⟩ → **dbl**. We see that the hybrid approach greatly increases the number of functions with no union of arrows at the function level, where 92% of functions only have one or two top-level alternatives. Also, nearly 5% of function types obtained with the basic union of arrows strategy have over 9 top-level alternatives.

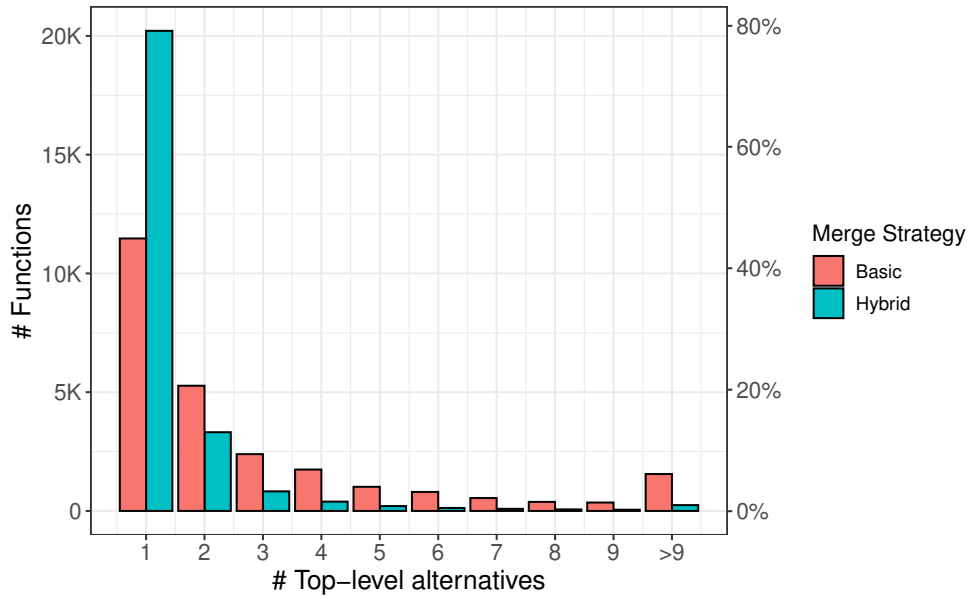


Fig. 9. Comparison of top-level function type counts across different merge strategies.

To connect this discussion with actual R types, we show what the types of the `correlation` function from the `agricolae` would be when using each of these strategies. `agricolae::correlation` obtains the coefficients of correlation and p-value between all variables of some input data table, using a method of the user's choosing. It returns the correlation matrix and the p-value (a double) together in a list. We refer the reader to Figure 10, where we see that the hybrid approach produces a much smaller type, with only two arrow types in the top-level union. Moreover, we see that no

real precision is lost when further reducing the function type to a pure arrow of unions, as the two top-level alternatives in the type obtained with the hybrid approach are not very informative.

Table 10. Types of `agricolae::correlation` with different type merge strategies.

Merge Strategy	Function Type
Union of Arrows	⟨**dbl**[], **dbl**[], **chr**, **chr**⟩ → **null** | ⟨**int**[], **dbl**[], **chr**, **chr**⟩ → **null** | ⟨**class**⟨data.frame⟩, **null**, **chr**[], **chr**⟩ → **list**⟨**class**⟨matrix⟩ | **dbl**⟩ | ⟨**class**⟨data.frame⟩, **null**, **chr**, **chr**⟩ → **list**⟨**class**⟨matrix⟩ | **dbl**⟩ | ⟨**dbl**[], **class**⟨data.frame⟩, **chr**[], **chr**⟩ → **list**⟨**class**⟨matrix⟩ | **dbl**⟩ | ⟨**dbl**[], **class**⟨data.frame⟩, **chr**, **chr**⟩ → **list**⟨**class**⟨matrix⟩ | **dbl**⟩
Hybrid	⟨**dbl**[], **dbl**[], **chr**, **chr**⟩ → **null** | ⟨**class**⟨data.frame⟩ | **dbl**[], ? **class**⟨data.frame⟩, **chr**[], **chr**⟩ → **list**⟨**class**⟨matrix⟩ | **dbl**⟩
Arrow of Unions	⟨**class**⟨data.frame⟩ | **dbl**[], ? **dbl**[] | **class**⟨data.frame⟩, **chr**[], **chr**⟩ → ? **list**⟨**class**⟨matrix⟩ | **dbl**⟩

In contrast, there are real cases where a meaningful loss of precision occurs. Consider instead the `rename` function from the `dplyr` package, with types shown in Figure 11. This function takes a data frame and renames selected columns. We see that the types are the same with both the union of arrows and hybrid approach (with only three unique function signatures observed), and can glean from these types that `dplyr::rename` produces a data frame of the *same class* as the input. This information is lost in the arrow-of-unions merge strategy. That said, once the type language is extend with proper user-defined type hierarchies, the arrow of unions type will be much smaller and more informative (as data frames, grouped tibbles, and tibbles are all related types).

Table 11. Types of `dplyr::rename` with different type merge strategies.

Merge Strategy	Function Type
Union of Arrows	⟨**class**⟨data.frame⟩, ...⟩ → **class**⟨data.frame⟩ | ⟨**class**⟨data.frame, grouped_df, tbl, tbl_df⟩, ...⟩ → **class**⟨data.frame, grouped_df, tbl, tbl_df⟩ | ⟨**class**⟨data.frame, tbl, tbl_df⟩, ...⟩ → **class**⟨data.frame, tbl, tbl_df⟩
Hybrid	⟨**class**⟨data.frame⟩, ...⟩ → **class**⟨data.frame⟩ | ⟨**class**⟨data.frame, grouped_df, tbl, tbl_df⟩, ...⟩ → **class**⟨data.frame, grouped_df, tbl, tbl_df⟩ | ⟨**class**⟨data.frame, tbl, tbl_df⟩, ...⟩ → **class**⟨data.frame, tbl, tbl_df⟩
Arrow of Unions	⟨**class**⟨data.frame⟩ | **class**⟨data.frame, tbl, tbl_df⟩ | **class**⟨data.frame, grouped_df, tbl, tbl_df⟩, ...⟩ → **class**⟨data.frame⟩ | **class**⟨data.frame, tbl, tbl_df⟩ | **class**⟨data.frame, grouped_df, tbl, tbl_df⟩

Finally, we want to briefly discuss types for base package functions (in R, functions like `+`, `-`, and vector access are part of the *base package*). These functions coerce mismatched arguments in an ad hoc manner, with no real defined semantics. Further, functions like `+` often act on values based on their type (according to `typeof`), and ignore the class, unless a package extended `+` to dispatch on the class. This leads to incredibly large inferred signatures for these functions. Even after counting out traces related to S3 and S4 dispatch, the type of `+` using the hybrid approach has 29 top-level alternatives, 22 of which have a class-typed argument. We don't believe this type to be particularly useful to a programmer, but a compiler might find it quite useful, as it can determine the type of the return value of a function given the types of its arguments.

8 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, 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.

ACKNOWLEDGMENTS

This work has received funding from the Office of Naval Research (ONR) award 503353, the National Science Foundation awards 1759736, 1544542, 1925644, and 1910850, the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412. This work was partially funded by NSERC.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385997

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.

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

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

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-14107-2_5

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 *Workshop on Dynamic Languages and Applications (DyLa)*. 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)*.

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

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1706299.1706343

A.3 Scala Implicits are Everywhere: A Large-scale Study of the Use of Scala Implicits in the Wild

Scala Implicits are Everywhere: A Large-scale Study of the Use of Scala Implicits in the Wild by *Filip Křikava, Heather Miller and Jan Vitek* published in *Proceedings of the ACM Programming Languages 3, OOPSLA*, Article 163, October 2019 [Křikava et al., 2019].

Scala Implicits Are Everywhere

A Large-Scale Study of the Use of Scala Implicits in the Wild

FILIP KŘIKAVA, Czech Technical University in Prague, CZ
HEATHER MILLER, Carnegie Mellon University, USA
JAN VITEK, Czech Technical University in Prague, CZ and Northeastern University, USA

The Scala programming language offers two distinctive language features *implicit parameters* and *implicit conversions*, often referred together as *implicits*. Announced without fanfare in 2004, implicits have quickly grown to become a widely and pervasively used feature of the language. They provide a way to reduce the boilerplate code in Scala programs. They are also used to implement certain language features without having to modify the compiler. We report on a large-scale study of the use of implicits in the wild. For this, we analyzed 7,280 Scala projects hosted on GitHub, spanning over 8.1M call sites involving implicits and 370.7K implicit declarations across 18.7M lines of Scala code.

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

Additional Key Words and Phrases: Implicit parameters, implicit conversions, corpora analysis, Scala

ACM Reference Format:
Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala Implicits Are Everywhere: A Large-Scale Study of the Use of Scala Implicits in the Wild. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 163 (October 2019), 28 pages. https://doi.org/10.1145/3360589

1 INTRODUCTION

"...experienced users claim that code bases are train wrecks because of overzealous use of implicits." –M. Odersky, 2017

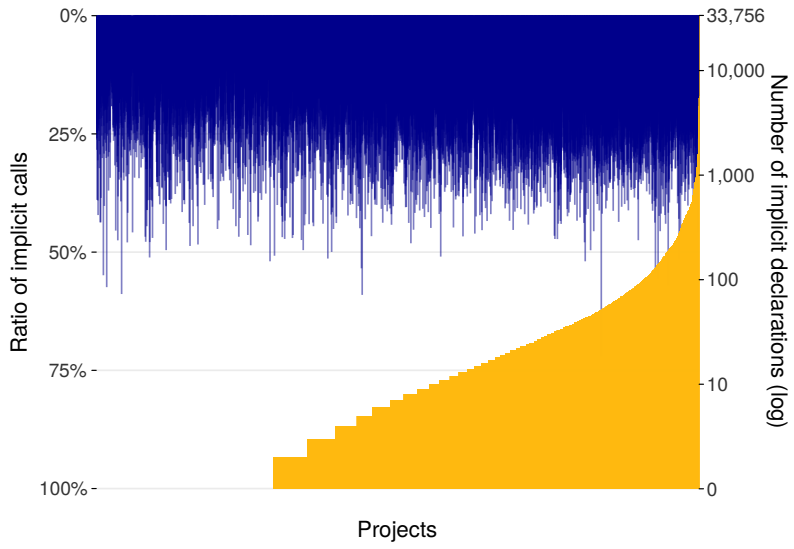
"...can impair readability or introduce surprising behavior, because of a subtle chain of inference." –A. Turon, 2017

"Any sufficiently advanced technology is indistinguishable from magic." –A.C. Clarke, 1962

Programming language designers strive to find ways for their users to express programming tasks in ways that are both concise and readable. One approach to reduce boilerplate code is to lean on the compiler and its knowledge and understanding of the program to fill in the "boring parts" of the code. The idea of having the compiler automatically provide missing arguments to a function call was first explored by Lewis et al. [2000] in Haskell and later popularized by Scala as *implicit parameters*. *Implicit conversions* are related, as they rely on the compiler to automatically adapt data structures in order to avoid cumbersome explicit calls to constructors. For example, consider the following code snippet: `"Just like magic!".enEspanol` Without additional context one would expect the code not to compile as the `String` class does not have a method `enEspanol`. In Scala, if the compiler is able to find a method to convert a string object to an instance of a class that has the required method (which resolves the type error), that conversion will be inserted silently by the compiler and, at runtime, the method will be invoked to return a value, perhaps `"Como por arte de magia!"`.

Implicit parameters and conversions provide ways to (1) extend existing software [Lämmel and Ostermann 2006] and implement language features outside of the compiler [Miller et al. 2013], and (2) allow end-users to write code with less boilerplate [Haoyi 2016]. They offload the task

2475-1421/2019/10-ART163
https://doi.org/10.1145/3360589



At a glance:

- 7,280 Scala projects
- 18.7M lines of code
- 8.1M implicit call sites
- 370.7K implicit declarations

98% of projects use implicits
78% of projects define implicits
27% of call sites use implicits

The top of the graph shows the ratio of call sites, in each project, that involves implicit resolution. The bottom shows the number of implicit definitions in each project.

Fig. 1. Implicits usage across our corpus

of selecting and passing arguments to functions and converting between types to the compiler. For example, the `enEspanol` method from above uses an implicit parameter to get a reference to a service that can do the translation: `def enEspanol(implicit ts:Translator):String`. Calling a function that has implicit arguments results in the omitted arguments being filled from the context of the call based on their types. Similarly, with an implicit conversion in scope, one can seamlessly pass around types that would have to be otherwise converted by the programmer.

The Good: A Powerful Tool. It is uncontroversial to assert that implicits changed how Scala is used. Implicits gave rise to new coding idioms and patterns, such as type classes [Oliveira et al. 2010]. They are one of a few key features which enable embedding Domain-Specific Languages (DSLs) in Scala. They can be used to establish or pass context (e.g., implicit reuse of the same threadpool in some scope), or for dependency injection. Implicits have even been used for computing new types and proving relationships between them [Miller et al. 2014; Sabin 2019]. The Scala community adopted implicits enthusiastically and uses them to solve a host of problems. Some solutions gained popularity and become part of the unofficial programming lexicon. As usage grew, the community endeavored to document and teach these idioms and patterns by means of blog posts [Haoyi 2016], talks [Odersky 2017] and the official documentation [Suereth 2013]. While these idioms are believed to be in widespread use, there is no hard data on their adoption. How widespread is this language feature? And what do people do with implicits? Much of our knowledge is folklore based on a handful of popular libraries and discussion on various shared forums.

Our goal is to document, for language designers and software engineers, how this feature is really used in the wild, using a large-scale corpus of real-world programs. We provide data on how they are used in popular projects engineered by expert programmers as well as in projects that are likely more representative of how the majority of developers use the language. This paper is both a retrospective on the result of introducing this feature into the wild, as well as a means to inform designers of future language of how people use and misuse implicits.

The Bad: Performance. While powerful, implicits aren't without flaws. Implicits have been observed to affect compile-time performance; sometimes significantly. For example, a popular Scala project reported a three order-of-magnitude speed-up when developers realized that an implicit conversion was silently converting Scala collections to Java collections only to perform a single

```
case class Card(n:Int, suit:String) {
  def isInDeck(implicit deck: List[Card]) =
    deck contains this
}
implicit def intToCard(n:Int) = Card(n, "club")
implicit val deck = List(Card(1, "club"))

1.isInDeck
```

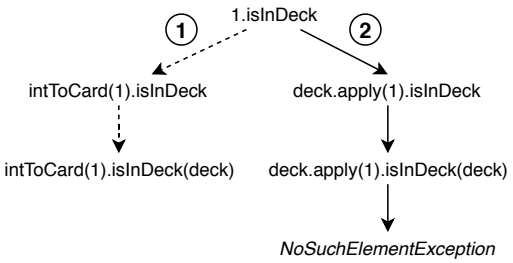


Fig. 2. Instead of injecting a conversion to `intToCard` (1), the compiler injects `deck.apply` (2) since `List[A]` extends (transitively) `Function[Int,A]`. An exception is thrown because the `deck` contains only one element (http://scalapuzzlers.com/)

operation that should have been done on the original object.[1] Another project reported a 56 line file taking 5 seconds to compile because of implicit resolution. Changing one line of code to remove an implicits, improved compile time to a tenth of second [Torreborre 2017]. Meanwhile, faster compilation is the most wished-for improvement for future releases of Scala [Lightbend 2018]. Could implicit resolution be a significant factor affecting compilation times across the Scala ecosystem?

The Ugly: Readability. Anecdotally, there are signs that the design of implicits can lead to confusing scenarios or difficult-to-understand code. Figure 2 illustrates how understanding implicit-heavy code can place an unreasonable burden on programmers[2]. In this example, the derivation chosen by the compiler leads to an error which requires understanding multiple levels of the type hierarchy of the `List` class. Such readability issues have even lead the Scala creators to reconsider the design of Scala's API-generation tool, Scaladoc. This was due to community backlash [Marshall 2009] following the introduction of the Scala 2.8 Collections library [Odersky and Moors 2009]—a design which made heavy use of implicits in an effort to reduce code duplication. The design caused a proliferation of complex method signatures across common data types throughout the Scala standard library, such as the following implementation of the `map` method which was displayed by Scaladoc as: `def map[B,That](f:A=>B)(implicit bf:CanBuildFrom[Repr,B,That]):That.` To remedy this, Scaladoc was updated with *use-cases*,[3] a feature designed to allow library authors to manually override method signatures with simpler ones in the interest of hiding complex type signatures often further complicated by implicits. The same `map` signature thus appears as follows in Scaladoc after simplification with a `@usecase` annotation: `def map[B](f: (A) => B): List[B]`

This Work. To understand the use of implicits across the Scala ecosystem, we have built an open source and reusable pipeline to automate the analysis of large Scala code bases, compute statistics and visualize results. We acquired and processed a corpus of 7,280 projects from GitHub with over 8.1M implicit call sites and more than 370.7K implicit declarations across 18.7M non-empty lines of Scala code. We observed over 98.2% projects using implicits, and 78.2% projects declaring implicits. With close to 27.2% of call sites requiring implicit resolution, implicits are the most used feature of Scala. Figure 1 summarizes the usage of implicits in our corpus. Our results document which idioms and patterns are popular and in application, library and tests. We provide data on the compilation time cost of implicits and the complexity of implicits. Our artifact is available at:

https://doi.org/10.5281/zenodo.3369436

[1] Documented in https://github.com/mesosphere/marathon/commit/fbf7f29468bda2ec29b7fbf80b6864f46a825b7a.
[2] For example, an entire book is devoted to so-called "puzzlers," or "enigmatic Scala code that behave highly contrary to expectations" which "will entertain and enlighten even the most accomplished developer" [Phillips and Serifovic 2014]
[3] *cf.* https://docs.scala-lang.org/overviews/scaladoc/for-library-authors.html

2 AN OVERVIEW OF SCALA IMPLICITS

Scala is a statically typed language that bridges the gap between object-oriented and functional programming. Implicits were included in the first release in 2004. In that version *implicit conversions* were used to solve the late extension problem; namely, given a class C and a trait T, how to have C extend T without touching or recompiling C. Conversions add a wrapper when a member of T is requested from an instance of C. Scala 2.0 added *implicit parameters* in 2006.

2.1 Implicit Conversions

Implicit conversion provides a way to use a type where another type is required without resorting to an explicit conversion. They are applied when an expression does not conform to the type expected by its context or when a called method is not defined on the receiver type. A conversion is defined with an implicit function or class, or an implicit value of a function type (*e.g.*, `implicit val x:A=>B`).

Implicit conversions are not specific to Scala. They also appear in languages such as C++ or C#. The difference is that conversions are typically defined in the class participating in the conversion, while in Scala the implicit conversions can be defined in types unrelated to the conversion types. This allows programmers to selectively import conversions. For example it is possible to define an implicit conversion from a `String` to an `Int`:

```
implicit def string2int(a: String): Int = Integer.parseInt(a)
val x: Int = "2"
```

Implicit conversions are essential to provide seamless interoperability with Java which was important in the early days of Scala. Conversions are also one of the main building blocks for constructing embedded Domain-Specific Languages (DSLs). For example, the following code snippet adds some simple time unit arithmetic that feels natural in the language.

```
case class Duration(time: Long, unit: TimeUnit) {
  def +(o: Duration) = Duration(time + unit.convert(o.time, o.unit), unit)
}
implicit class Int2Duration(that: Int) {
  def seconds = new Duration(that, SECONDS); def minutes = new Duration(that, MINUTES)
}
5.seconds + 2.minutes //Duration(125L, SECONDS )
```

2.2 Implicit Parameters

A method or a constructor can define implicit parameters. The arguments to these parameters will be filled in by the compiler at every call site with the most suitable values in the calling context. For example, a function `def sub(x:Int)(implicit y:Int)=x-y` with implicit parameter `y` can be called with `sub(1)` provided that the compiler can find an implicit such as `implicit val n=1`. The compiler looks for implicits in the current lexical scope and if there are no eligible identifiers then it searches the implicit scope of the implicit parameter's type (associated companion objects[4] and packages). If a value is found, the compiler injects it into the argument list of the call. If multiple values are found and none of them is more specific than the others, an ambiguity compilation error is thrown. An error is also raised when no eligible candidate is found. Importantly, besides having the correct type, an implicit value from a lexical scope must be accessible using its simple name (without selecting from another value using dotted syntax). This means that normal rules for name binding including shadowing apply. Implicit values (`val`), variables (`var`), objects (`object`) and functions (`def`) without explicit parameters can all be used to fill implicit parameters. An implicit parameter of a function type `A => B` can be used as an implicit conversion in the method body. For example

[4] A companion object is a singleton associated with a class used to define static fields and methods.

`def get[T, C](xs: C, n: Int)(implicit conv: C => Seq[T]):T = xs(n)` can be called with any type `C`, as long as there is implicit conversion in scope that can convert `C` into a sequence.

2.3 Idioms and Patterns

Over time, programmers have put implicits to many uses. This section describes the most widely discussed implicit idioms. This list is based on our understanding of the state of practice. It is not expected to be exhaustive or definitive.

2.3.1 Late Trait Implementation. This idiom is a solution for the late extension problem, and was the original motivation for adding implicits to Scala in the first place. To add a new trait to an existing class, one can define a one-parameter conversion that returns an instance of the trait.

```
implicit def call2Run(x: Callable[_]): Runnable = new Runnable { def run = x.call }
```

This snippet adds the `Runnable` interface to any any type that implements `Callable`. Conversions can also take implicit parameters, they are then referred to as *conditional* conversions.

```
implicit def call2Future[T](x: Callable[T])(implicit ctx: ExecutionContext): Future[T]
```

For example, the above defines a late trait implementation that is only applicable if there exists an execution context in scope.

2.3.2 Extension Methods. Extension methods allow developers to add methods to existing classes. They are defined with an `implicit def` that converts objects to a new class that contains the desired methods. Scala 2.10 added syntactic sugar to combine conversion and class declaration in the `implicit class` construct. The conversion takes a single non-implicit parameter as shown in the following snippet where `zip` is added to any `Callable`.

```
implicit class XtensionCallable[T](x: Callable[T]) {
  def zip[U](y: Callable[U]): Callable[(T, U)] = () => (x.call, y.call)
}
val c1 = () => 1; val c2 = () => true; val r = c1 zip c2 // r: Callable[(Int, Boolean)]
```

An extension method is convenient as it allows to write `c1 zip c2` instead of `zip(c1, c2)`. It is an important feature for embedded DSLs. On the other hand, unlike static methods, it is harder to read. Without knowing the complete code base it is difficult to know where a calling method is defined and how the definition got into the current scope. Extension methods can also be conditional. For example, we can add a `def schedule(implicit c: ExecutionContext)` method that will run the callable on the implicitly provided execution context if it is present. If there is none, the developer will get a compile-time error "*cannot find an implicit ExecutionContext ... import scala.concurrent.ExecutionContext.global.*". This is because the `ExecutionContext` is annotated with `@implicitNotFound`, a Scala annotation allowing one to customize the compile-time error message that should be outputted in the case no implicit value of the annotated type is available.

2.3.3 Type Classes. Oliveira et al. [2010] demonstrated how to use implicit parameters to implement type classes [Wadler and Blott 1989]. Fig. 3a defines a trait `Show` that abstracts over pretty-printing class instances. The function `show` can be called on instances `T`, for which there is an implicit value of type `Show[T]`. This allows us to retrospectively add support to classes we cannot modify. For example, given a class `Shape(sides: Int)` from a 3rd party library, we can define the implicit value `ShapeShow` to add pretty printing (Fig. 3b). This is an implicit object that extends `Show` and implements `show`. Thus when `show` is called with an explicit argument of type `Shape`, for example `show(Shape(5))`, the compiler adds the implicit `ShapeShow` as the implicit argument `ev`, resulting in `show(Shape(5))(shapeShow)`.
Since functions can be used as implicit parameters, we can generalize this example and create an implicit allowing us to show a sequence of showable instances. In the following snippet, `listShow` is

```scala
trait Show[T] {
  def show(x: T): String
}

def show[T](x: T)(implicit ev: Show[T]) =
   ev.show(x)
```

(a)

```scala
case class Shape(n: Int)

implicit object shapeShow extends Show[Shape] {
  def show(x: Shape) = x.n match {
    case 3 => "a triangle"; case 4 => "a square"
    case _ => "a shape with $n sides" }
  }
```

(b)

```scala
implicit def listShow[T](implicit ev: Show[T]) = new Show[List[T]] {
  def show(x: List[T]) = x.map(x => ev.show(x)).mkString("a list of [", ", ", "]")
}
```

(c)

Fig. 3. Type classes

a generic type class instance that combined with an instance of type `Show[T]` returns a type class instance of type `Show[List[T]]` (Fig. 3c). Thus, a call to `show(List(Shape(3), Shape(4)))` is transformed to `show(List(Shape(3), Shape(4)))(listShow[Shape](shapeShow))`, with two levels of implicits inserted. This implicit type class derivation is what makes type classes very powerful. The mechanism can be further generalized using implicit macros to define a *default* implementation for type class instances that do not provide their own specific ones [Miller et al. 2014; Sabin 2019].

2.3.4 Extension Syntax Methods. Type classes define operations on types, when combined with extension methods it is possible to bring these operations into the corresponding model types. We can extend the `Show[T]` type class and define an extension method

```scala
implicit class ShowOps[T](x: T)(implicit s: Show[T]) { def show = s.show(x) }
```

allowing one to write directly `Shape(3).show` instead of `show(Shape(3))`. The `ShowOps[T]` is a conditional conversion that is only applied if there is an instance of the `Show[T]` in scope. This allows library designers to use type class hierarchies instead of the regular sub-typing. The name *extension syntax methods* comes from the fact that developers often lump these methods into a package called *syntax*.

2.3.5 Type Proofs. Implicit type parameters can used to enforce API rules at a compile time by encoding them in types of implicit parameters. For example, `flatten` is a method of `List[A]` such that given an instance `xs: List[List[B]]`, `xs.flatten` returns `List[B]` concatenating the nested lists into a single one. This is done with an implicit parameter:

```scala
class List[A] { def flatten[B](implicit ev: A => List[B]): List[B] }
```

Here, `A => List[B]` is an implicit conversion from `A` to `List[B]`. It can also be viewed as a predicate that must be satisfied at compile time in order for this method to be called. We can define an implicit function `implicit def isEq[A]: A=>A = new =>[A,A]{}` that will act as generator of proofs such that `A` in `A => List[B]` is indeed `List[B]`. Therefore, a call `List(List(1)).flatten` will be expanded to `List(List(1)).flatten(isEq[List[Int]])` since `A` is a `List` while `List(1).flatten` will throw a compile time exception: *"No implicit view available from Int => List[B]"*.

2.3.6 Contexts. Implicit parameters can reduce the boilerplate of threading a context parameter through a sequence of calls. For example, the methods in `scala.concurrent`, the concurrency library in Scala's standard library, all need an `ExecutionContext` (e.g., a thread pool or event loop) to execute

their tasks upon. The following code shows the difference between explicit and implicit contexts.

```
val ctx = ExecutionContext.global
val f1 = Future(1)(ctx)
val f2 = Future(2)(ctx)
val r = f1.flatMap(r1 =>
  f2.map(r2 => r1 + r2)(ctx)
)(ctx)
```

With *explicit* context

```
implicit val ctx = ExecutionContext.global
val f1 = Future(1)
val f2 = Future(2)


val r = for(r1 <- f1; r2 <- f2) yield r1 + r2
```

With *implicit* context

On the left, an explicit context is passed around on every call to a method on `Future`, while on the right much of the clutter is gone thanks to implicits. This de-cluttering hides the parameters and makes calls to `map` and `flatMap` more concise. The idiom consists of the declaration of an implicit context (usually as an `implicit val`), and the declaration of the functions that handle it.

2.3.7 Anti-patterns: Conversions. A widely discussed anti-pattern is the conversions between types in unrelated parts of the type hierarchy. The perceived danger is that any type can be automatically coerced to a random type unexpectedly; *e.g.,* imagine a conversion from `Any` to `Int` introduced into the root of a big project. One could imagine such a conversion wreaking havoc in surprising places in a code base and being difficult to track down. Another anti-pattern is conversions that go both ways [Odersky 2017]. Since conversions are not visible, it is difficult to reason about types at a given call site as some unexpected conversion could have happened. An example is the, now deprecated, Java collection conversion. In an earlier iteration, Scala defined implicit conversions between Java collections and its own, such as:

```
implicit def asJavaCollection[A](it: Iterable[A]): java.util.Collection[A]
implicit def collectionAsScalaIterable[A](i: java.util.Collection[A]): Iterable[A]
```

As they were often imported together using a wildcard import `collection.JavaConversions._`, it was easy to mistakenly invoke a Java method on a Scala collection and vice-versa silently converting the collections from one to another. Furthermore, in this case, these conversions also change semantics as the notion of equality in Java collections is different from Scala collections (reference vs. element equality). Since implicit conversions can introduce some pitfalls, the compiler issues a warning when compiling an implicit conversion definition. It can be suppressed by an import (or a compiler flag) which is usually automatically done by an IDE and thus diminishing the utility of these warnings.

2.4 Complexity

Implicits help programmers by hiding the "boring parts" of programming tasks, the plumbing that does not require skill or attention. The problem is that, as the above idioms demonstrate, implicits are also used for subtle tasks. Their benefits can turn into drawbacks. One way to measure the potential complexity of implicits is to look at the work done by the compiler. When implicits work, programmers need not notice their presence. But when an error occurs, the programmer suddenly has to understand the code added by `scalac`. For example, a comparison of two tuples `(0,1)<(1,2)` gets expanded to `orderingToOrdered((0,1))(Tuple2(Int, Int))<(1,2)`. The compiler injects two additional calls (`orderingToOrdered` implicit conversion, `Tuple2` type class) with two implicit arguments (`Int`). The question is how much of this *filling* there is.

Tooling can help navigate the complexity added by implicits. The plugin for IntelliJ IDEA has a feature that can show implicit hints, including the implicit resolution in the code editor. This effectively reveals the injected code making it an indispensable tool for debugging. However, turning the implicit hints on severely hinders the editor performance, creating a significant lag when working

with implicits-heavy files. The second problem with this is that the IntelliJ compiler is not the same as `scalac`, and often implicit resolution disagrees between the two compiler implementations.

Another common problem that hinders understanding is related to implicit resolution. Eligible implicits for both conversions and parameters are searched in two different scopes. The search starts in the lexical scope that includes local names, enclosing members and imported members and continues in the implicit scope that consists of all companion objects associated with the type of the implicit parameter. The advantage of the implicit scope is that it does not need to be explicitly imported. This prevents errors caused by missing imports for which, due to the lack of global implicit coherence, the compiler cannot give a better error message than a type mismatch, *"member not found"* or *"could not find implicit value"*. Implicit scope has a lower priority allowing users to override defaults by an explicitly importing implicit definition into the lexical scope. A consequence of this is that an import statement can change program semantics. For example, in the code bellow contains two late trait implementations of a trait `T` for a class `A`: `C1` defined in implicit scope of the class `A`, and `C2` defined in an unrelated object `O`:

```
trait T { def f: Int }
class A; object A { implicit class C1(a: A) extends T { def f = 1 } }
object O { implicit class C2(a: A) extends T { def f = 2 } }
new A().f
```

At the call to `f`, the compiler will use the `C1` conversion resolved from the implicit scope so the result will be `1`. However, if later there is an `import O._` before the call site, the expression will return `2`. The import will bring `C2` into the lexical scope prioritizing `C2` over `C1`.

Further, implicits defined in the lexical scope follow the name binding rules and thus can be shadowed by explicit local definitions. For instance, adding any definition with a name `C2` (*e.g.,* `val C2 = null`) into the scope before the call to `f` will result in returning again `1`, since the imported `O.C2` implicit will be shadowed by this local definition. In the case `C1` did not exits, the compiler will simply emit *"value f is not a member of A"* error. To avoid this, library authors try to obfuscate the implicits names which in turn affects the ergonomics. A notable example is in the Scala standard library where the implicit providing a proof that two types are in a sub-type relationship is named `$conforms` in order to prevent a potential shadowing with a locally defined `conforms` method.[5]

2.5 Overheads

Implicit resolution together with macro expansion can sometimes significantly increase compilation time. To illustrate the problem, consider the JSON serialization of algebraic data types using the `circe.io`[6], a popular JSON serialization library. We define two ADTs: `case class A(x: String)` and `case class B(xs: List[A], ys: List[A])`, and a method to print out their JSON representation:

```
def print(a: A, b: B) = println(a.asJson, b.asJson)
```

The `asJson` method is an extension method defined in the `circe.io` as `def asJson(implicit encoder: Encoder[T]): Json`. It uses an implicit parameter of type `Encoder[T]` effectively limiting its applicability to instances that define corresponding encoder. For the code to compile, two encoders `Encoder[A]` and `Encode[B]` that turn `A` and `B` into `Json` are needed. The `circe.io` library gives three options for creating the encoder: manual, semi automated and automated.

The manual encoding involves implementing the single method in `Encoder`, manually creating an instance of `Json` with the appropriate fields (*cf.* Listing. 4a). While simple, it is a boilerplate code. The semi-automated solution delegates to `derivedEncored` that synthesizes the appropriate type at compile time through implicit type class derivation and macros (*cf.* Listing. 4b). The fully

[5] Reported in Scala issue #7788, *cf.* https://github.com/scala/bug/issues/7788
[6] *cf.* https://github.com/circe/circe

```scala
object manual {
  implicit val eA: Encoder[A] = (a: A) =>
   obj("x"->str(a.x))
  implicit val eB: Encoder[B] = (b: B) =>
   obj("xs" -> arr(b.xs.map(_.asJson)),
       "ys" -> arr(b.ys.map(_.asJson)))
}
```

(a) manual

```scala
object semiauto {
  import io.circe.generic.semiauto._

  implicit val eA: Encoder[A] = deriveEncoder[A]
  implicit val eB: Encoder[B] = deriveEncoder[B]
}
```

(b) semi-automated

```scala
import io.circe.generic.auto._
```

(c) automated

Fig. 4. Type class derivations

automated solution, does not require extra code at the client side beside importing its machinery (*cf.* Listing. 4c). Compile time is affected by the choice of approach; taking the manual as a base line, semi-automated is 2.5x slower and automated is 3.8x slower.

The reason for this compile-time slow-down is the increase in the number of implicit resolutions triggered and macro expansion as shown in Table 1. The difference between the automated and semi-automated is that the former *caches* the derived instances in the implicit values `eA` and `eB` and so the `eA` which is synthesized before `eB` will be reused for deriving `eB`. The automated derivation synthesizes new instances for each application. In this simple example, it generates 140 additional lines of code at the `println`[7]. Caching of derived type classes was already reported to significantly improve the compilation time of various projects [Cantero 2018; Torreborre 2017]. One difficulty is that since the implicit scope is *invisible*, it is harder to figure out which implicits are derived where and which are causing slowdowns. Currently, the only way is to use a *scalac-profiling*[8], compiler plugin which outputs more detailed statistics about implicit resolution and macro expansion.

Table 1. Count of implicit resolutions and macro expansions, and timing of the typer phase in `scalac` 2.12.8 with `-Ystatistics:typer` flag.

	Implicits	Macros	Time
Manual	13	0	.1s
Semi	35	51	.3s
Auto	52	78	.5s

3 SCALA ANALYSIS PIPELINE

We have implemented a data analysis pipeline targeting large-scale analysis of Scala programs. To the best of our knowledge, this is the only pipeline able to scale to thousands of projects. Our infrastructure can be extended for other analyses and it is available in open source.

Figure 5 gives an overview of the pipeline; every step shown in the figure is fully automated. The first step is to download projects hosted on GitHub. Next, gather basic metadata and in particular infer the build system each project uses. Incompatible projects are discarded in the next step. These are projects that do not meet the technical requirements of the analysis tools. The fourth step is to use the DéjàVu tool [Lopes et al. 2017] to filter out duplicate projects. The fifth step is to attempt to compile the corpus and generate semantic information. The final step is to load the extracted data and analyze them. The pipeline is run in parallel using GNU parallel [Tange et al. 2011] but the analysis is resource intensive. On our server (Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM) we were not able to compile more than 12 projects in parallel.

[7] Measured in the expanded code obtained from `-Xprint:typer` compiler flag
[8] *cf.* https://github.com/scalacenter/scalac-profiling

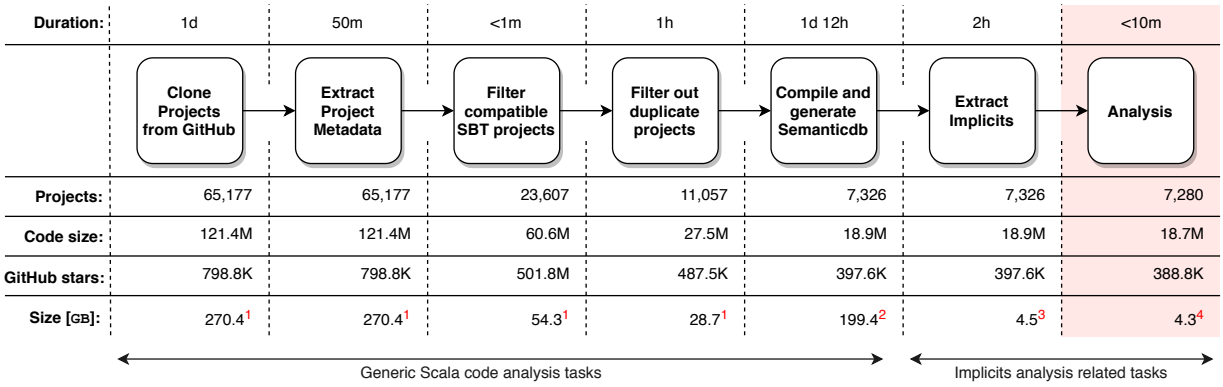


Fig. 5. Scala Analysis pipeline. (1) is the size of source code, (2) is the size of source plus compiled code and generated SemanticDB, (3) is the size of extracted implicits data model, (4) is the size of exported CSV files. The code size include tests.

The pipeline is reusable for other semantic analyses on Scala code bases, as only the last two steps relate specifically to implicits. At the end of the *Compile and generate SemanticDB* task, the corpus contains built projects with extracted metadata and SemanticDB files—these SemanticDB files contain syntactic information as well as semantic information (Scala symbols and types).

The pipeline logs all the steps for each project and provide an aggregated summaries. The analysis is done in R, and even though it is possible to load Google Protocol Buffers into R, it is not practical. Thus, we first aggregate the extracted data and export them into CSV format, which is more natural to work with in R. This is implemented in ~500 lines of make files and ~5K of R code. The implicit extractor is written in ~7.2K lines of Scala code.

The pipeline uses Scalameta[9], a library that provides a high-level API for analyzing programs. One part of this library is a compiler plugin that for each compilation unit produces a data model with syntactic and semantic information. This includes a list of defined and referenced symbols as well as synthetic call sites and parameters injected by the compiler. The result is stored in a binary SemanticDB[10] file (in Google Protocol Buffer serialization format). It can also extract symbol information from compiled classes allowing us to find implicits defined in external project dependencies. Note that this analysis would have not been possible with only syntactic information; compile-time information like types is required to match up call site and declaration site due to the fact that implicits themselves are type-directed rewritings performed by the compiler at type-checking time.

Based on this we have built a tool that extract implicit declarations and call sites. There are two limitations with Scalameta: it is limited to certain versions of Scala (2.11.11 in the 2.11 branch and 2.12.4 in the 2.12 branch), and it does not support *white-box macros* (*i.e.,* macros without precise signatures in the type system before their expansion) [Burmako 2017].

Another thing to consider when using SemanticDB is that it requires compiling the projects. The Scala compiler is about an order of magnitude slower than a Java compiler[11] and the SemanticDB compiler plugin adds additional overhead. For our analysis Sbt is used to rebuild each project three times. There is no easy way around this. As noted above, lightweight, syntax-based approaches using regular expressions or pattern matching over AST nodes would not work because the call

[9] *cf.* https://scalameta.org/
[10] *cf.* https://scalameta.org/docs/semanticdb/specification.html
[11] *cf.* https://stackoverflow.com/a/3612212/219584

sites that use implicits are not visible in the source/AST, and to identify these patterns requires resolving terms and types from the declaration- and use-sites.

Scala projects are compiled by build tools which are responsible for resolving external dependencies. We chose Sbt as it is the most-used tool in the Scala world. Since version 0.13.5 (August 2014), it supports custom plugins which we use to build an extractor of metadata. Next to the version information and source folder identification, the extracted metadata gives us information about project internal and external dependencies. This is necessary for assembling project's class-path that is used to resolve symbols defined outside of the project.

3.1 Implicit Extraction

The SemanticDB model contains low-level semantic information about each compilation unit. This includes synthetics, trees added by compilers that do not appear in the original source (*e.g.,* inferred type arguments, for-comprehension desugarings, `C(...)` to `C.apply(...)` desugarings, implicit parameters and call sites). These trees are defined as transformations of pieces of the original Scala AST and as such they use quotes of the original sources. For example, the following Scala code:

```
import ExecutionContext.global; Future(1)
```

will have two synthetic trees injected by the compiler:

```
- ApplyTree(OriginalTree(1,60,1,86), IdTree("EC.global"))
- TypeApplyTree(SelectTree(OriginalTree(1,60,1,83),IdTree("Future.apply()")),TypeRef("Int"))
```

In this form, SemanticDB is not convenient for higher-level queries about the use of implicits. In order to do this, we transform SemanticDB into our own model that has declarations and call sites resolved. This is done in two steps. First, we extract implicit declarations by traversing each compilation unit and collecting declarations with the `implicit` modifier. For each declaration, we resolve its type using the symbol information from the SemanticDB and the project class path. This is done recursively in the case the declaration type has parents. Next, we look into the synthetic trees and extract inserted implicit function applications. Together with the project metadata, both declaration and call sites are stored in a tree-like structure using the Google Protocol Buffer format. In our example, the extractor will produce 13 declarations and one implicit call site including:

```
  // def apply[T](body: => T)(implicit executor: EC)
- Declaration("Future.apply()", DEF, ret=Ref("Future.apply().[T]"), params=List(
    ParamList(Param("body", Ref("Future.apply().[T]"))),
    ParamList(Param("executor", Ref("EC"), isImplicit=true))))
  // implicit val global: EC
- Declaration("EC.global", VAL, ret=Ref("EC",List()), isImplicit=true)
  // Future.apply[Int](1)(EC.global)
- CallSite("Future.apply()", typeArgs=Ref("Int"), implicitArgs=Ref("EC.global"))
```

Such model can be queried using the standard Scala collection API. For example, we can list a project's `ExecutionContext` declarations and the corresponding call sites that use them as follows:

```
val declarations = proj.declarations filter (dcl =>
  dcl.isImplicit && dcl.isVal && dcl.returnType.isKindOf("EC"))
val callsites = {
  val ids = declarations.map(_.declarationId).toSet
  proj.implicitCallsites filter (cs =>
    cs.implicitArguments exists (arg => ids contains arg))
}
```

The extractor is run per project in parallel and the results are merged into one binary file. This file can be streamed into a number of processors that export information about declarations, call sites, implicit conversions and implicit parameters into CSV files.

4 PROJECT CORPUS

For this paper we analyzed 7,280 projects consisting of 18.7M lines of Scala code (including 5.9M lines of tests and 2.2M lines of generated code). Most projects are small, the median is 677 lines of code, but the corpus also includes projects with over 100K lines of source code. 4,197 projects use Scala 2.11 but they account for less code (43.8%) and fewer stars (33.7%). For the remainder of the paper we partition our corpus in four categories: **small apps** are project with fewer than 1,000 LOC, **large apps** are projects with more than 1,000 LOC, **libraries** are projects that are listed on Scaladex. We also extract the test code from all projects into the **tests** category. Scaladex is a package index of projects published in Maven Central and Bintray repositories. These labels are somewhat ad-hoc as there is not always a strong reason behind the addition of a project to Maven Central or Bintray. However, manual inspection suggests that most of the projects that appear on Scaladex are intended for reuse.

Table 2. Project categories

Category	Projects	Code size	GitHub stars	Commits
Small apps.	3.3K	1M (mean=0.3K)	28K (mean=8)	139K (mean=41)
Large apps.	1.3K	5M (mean=4.0K)	74K (mean=57)	425K (mean=325)
Libraries	2.6K	6M (mean=2.4K)	285K (mean=108)	712K (mean=271)
Tests	5.4K	5M (mean=1.1K)	-	-

Figure 6 shows all projects, the size of the dots reflects number of stars, the color their category (large/small apps or libraries), the x-axis indicates the number of lines of code (excluding 5.9M lines of tests) in log scale, the y-axis gives the number of commits to the project in log scale. Solid lines indicate the separation between small and large applications. Dotted lines indicate means.

The corpus was obtained from publicly available projects listed in the GHTorrent database [Gousios 2013] and Scaladex. The data was downloaded between January and March 2019. We started with 65,177 non-empty, non-fork projects, which together contained 121.4M lines of code. We filtered out projects that were not compatible with our analysis pipeline (e.g., projects using early versions of Scala) and removed duplicates. 43K use Sbt as their build system (other popular build systems are Maven with 5.1K projects and Graddle with 1.5K). From the Sbt projects, 23.6K use Sbt version 0.13.5+ or 1.0.0+ that is required by our analysis. We thus discarded about half of the downloaded code.

For duplicates, the problem is that even without GitHub forks, the corpus still contained unofficial forks, *i.e.,* copies of source code. For example, there were 102 copies of `spark`. Since `spark` is the largest Scala project (over 100K LOC), keeping them would significantly skew the subsequent analysis as 37.6% of the entire data set would be identical. In general, getting rid of duplicate projects is difficult task as one needs to determine the origins of individual files. We use that following criteria to retain a project: (1) it must have more than one commit, (2) it must be active for at least 2 months, (3) it must be in Scaladex or have less than 75% of file-level duplication or more than 5 stars on GitHub, and (4) it must be in Scaladex or have less than 80% duplication or more than 500 stars on GitHub. These rules were tuned to discard as many duplicates as possible while keeping originals. While large numbers of GitHub stars do not necessarily mean that a project widely-used, originals tend to have higher star counts than copies. The actual thresholds were chosen experimentally to make sure we keep all the bigger (> 50K LOC) popular Scala projects without any duplicates. We excluded 12,550 projects (33.1M lines of code). While this is over half of the source code from the compatible Sbt projects, we lost fewer than 2.8% stars.

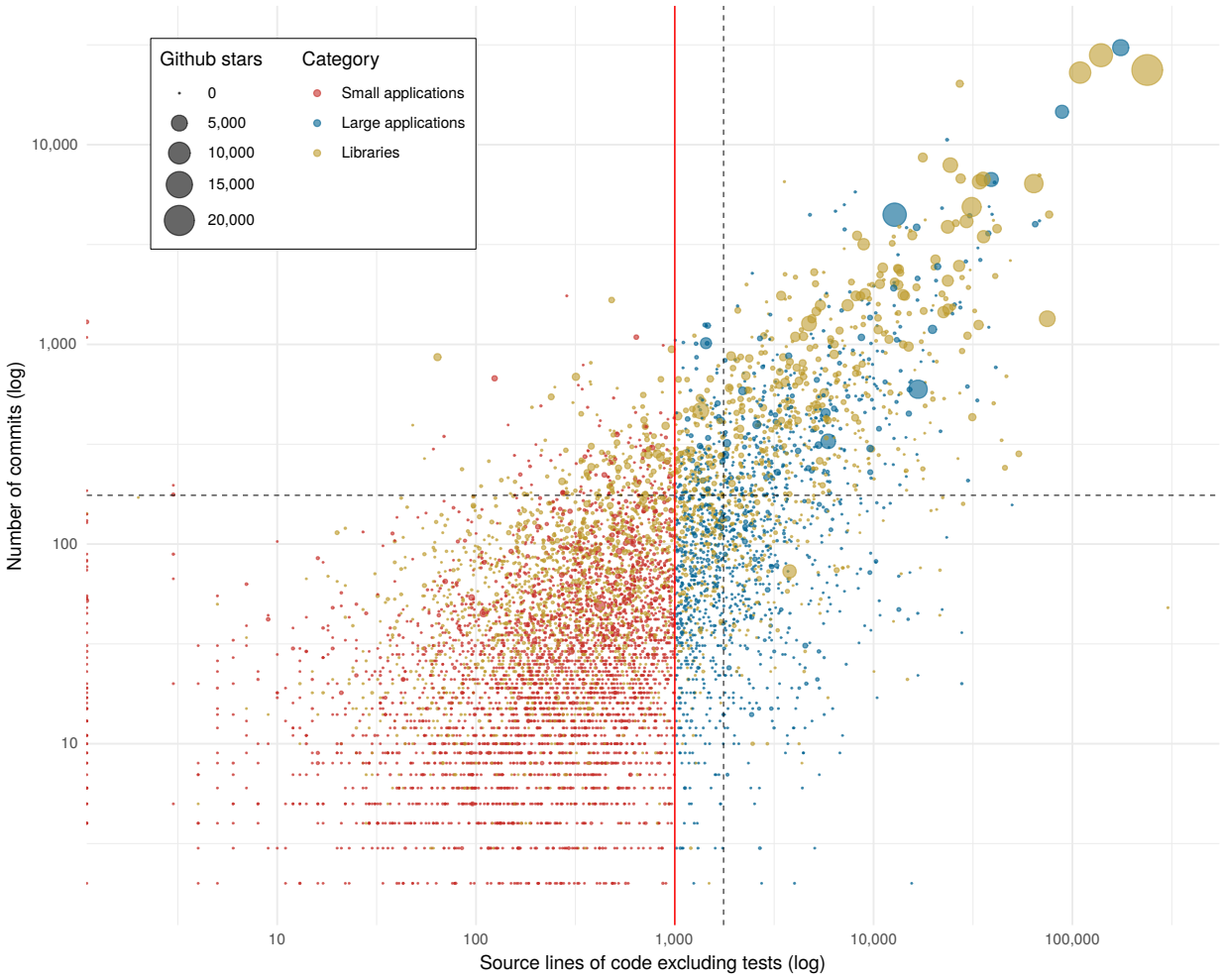


Fig. 6. Corpus overview

From the resulting 11,057 projects, we were able to successfully compile 7,326 projects. 3,731 projects failed to build. We follow the standard procedure of building Sbt projects. If a project required additional steps, we marked it as failed. The following are the main sources of failures:

– *Missing dependencies* (2.1K). Most missed dependencies were for `scalajs` (964), a Scala-to-JavaScript compiler with a version that was likely removed because of security vulnerabilities. The next most frequent issue was due to snapshot versions (263) that were no longer available. The remainder were libraries that were taken down or that reside in non-standard repositories. Following common practice, we use a local proxy that resolves dependencies. No additional resolvers were configured. The proxy downloaded 204K artifacts (110GB).
– *Compilation error* (873). Some commits do not compile, and others fail to compile due our restriction on Scala versions. Scalameta requires Scala 2.11.9+ or 2.12.4+. Some projects are sensitive even down to the path version number. Some of these version upgrades might have also caused the missing dependencies in case the required artifact was built for a particular Scala version.
– *Broken build* (189). The Sbt could not even start due to errors in the `build.sbt`.
– *Empty build* (156). Running Sbt did not produce class files, leaving the projects *empty*. This happens when the build has some non-standard structure.

Finally, in the analysis, we discarded 46 projects (1.1% of the code) because some of their referenced declarations were not resolvable (the Scalameta symbol table did not return any path entry) and inconsistencies in SemanticDB. Table 3 lists some of the top rated projects that were included in the final corpus, including number of stars, lines of code, number of commits, level of duplication, Scala version and whether it is listed in Scaladex.

Table 3. Top 40 open source projects

Project	GitHub stars	Code size	Commits	Duplication	Scala version	Scaladex
apache/spark	21,067	238,062	23,668	0.4	2.12.8	Y
apache/predictionio	11,696	12,764	4,461	0	2.11.12	N
scala/scala	11,386	139,300	28,062	0.9	2.12.5	Y
akka/akka	9,666	109,359	22,966	0.001	2.12.8	Y
gitbucket/gitbucket	7,612	31,144	4,874	0	2.12.8	Y
twitter/finagle	7,003	63,976	6,386	0.01	2.12.7	Y
yahoo/kafka-manager	6,958	16,733	596	0.5	2.11.8	N
ornicar/lila	5,218	175,054	30,617	0.01	2.11.12	N
rtyley/bfg-repo-cleaner	5,014	1,351	465	0	2.12.4	Y
linkerd/linkerd	4,910	74,775	1,344	0.003	2.12.1	Y
fpinscala/fpinscala	4,244	5,914	327	1	2.12.1	N
haifengl/smile	4,242	4,731	1,271	0	2.12.6	Y
gatling/gatling	4,151	24,322	7,900	0	2.12.8	Y
scalaz/scalaz	4,079	34,146	6,523	0	2.12.8	Y
mesosphere/marathon	3,823	39,097	6,694	0.03	2.12.7	N
sbt/sbt	3,782	35,574	6,726	0.4	2.12.8	Y
twitter/diffy	3,375	3,778	73	0	2.11.7	Y
lampepfl/dotty	3,278	88,680	14,616	0.3	2.12.8	N
twitter/scalding	3,113	29,346	4,133	0	2.11.12	Y
typelevel/cats	3,093	23,607	3,878	0.009	2.12.7	Y
scalanlp/breeze	2,816	35,747	3,461	0.002	2.12.1	Y
scalatra/scalatra	2,382	8,914	3,174	0.3	2.12.8	Y
netflix/atlas	2,288	22,474	1,450	0	2.12.8	Y
spark-jobserver/spark-jobserver	2,286	7,403	1,571	0.3	2.11.8	Y
twitter/util	2,243	26,927	2,472	0.2	2.12.7	Y
slick/slick	2,188	23,622	2,084	0	2.11.12	Y
laurilehmijoki/s3_website	2,178	1,435	1,014	0	2.11.7	N
twitter/summingbird	2,011	9,057	1,790	0.3	2.11.12	Y
MojoJolo/textteaser	1,942	420	49	0	2.11.2	N
twitter/finatra	1,888	14,071	1,772	0.001	2.12.6	Y
twitter/algebird	1,836	23,676	1,502	0	2.11.12	Y
scala-exercises/scala-exercises	1,775	5,398	1,570	0	2.11.11	Y
circe/circe	1,633	8,140	1,749	0.006	2.12.8	Y
datastax/spark-cassandra-connector	1,569	11,120	2,418	0.2	2.11.12	Y
rickynils/scalacheck	1,480	4,038	1,091	0	2.12.6	Y
monix/monix	1,466	33,749	1,251	0	2.12.8	Y
http4s/http4s	1,459	27,412	6,765	0.003	2.12.7	Y
sangria-graphql/sangria	1,442	14,999	975	0.2	2.12.7	Y
spotify/scio	1,439	20,477	2,659	0.002	2.12.8	Y
coursier/coursier	1,417	13,313	1,984	0	2.12.8	Y

5 ANALYZING IMPLICITS USAGE

This section presents the results of our analysis and paints a picture of the usage of implicits in our corpus of Scala programs. We follow the structure of Section 2 and give quantitative data on the various patterns and idioms we presented including details about how identified them. We further discuss the impact of implicits on code comprehension and compilation time.

Identifying implicits requires performing a number of queries on the data files produced by our pipeline. Doing this also turned out to be necessary to remove duplication due to compilation artifacts. These come from projects compiled for multiple platforms and projects compiled for multiple major versions of Scala. While the main compilation target for Scala projects is Java byte-code (7,075 projects), JavaScript and native code are also potential targets. To prevent double counting, we make sure that shared code is not duplicated. Since Scala 2.11 and 2.12 are not binary compatible, libraries supporting both branches cross compile to both versions. We take care to compile only to one version.

In the remainder of this paper, when we refer to the "Scala library," "Scala standard library," or sometimes just to "Scala" we mean code defined in `org.scala-lang:scala-library` artifact.

Overview of Results. Out of the 7,280 analyzed projects, 7,148 (98.2%) have at least one implicit call site. From over 29.6M call sites in the corpus (explicit and implicit combined), 8.1M are call sites involving implicits. Most of these calls are related to the use of implicit parameters (60.3%). Figure 7 shows for each category a distribution of implicit call site ratios. The box is the 25th/75th percentiles and the line inside the box represents the median with the added jitters showing the actual distribution. For applications and libraries, the median is similar. It is smaller ~17.1%. In the case of test code, it is more than double, 38%. There tend to be more implicit call sites in tests than in the rest of the code. That is not surprising because the most popular testing frameworks heavily rely on implicits. Across the project categories the median is 23.4% (shown by the dashed line)—*i.e., one out of every four call sites involves implicits.*

Figure 8 shows the distribution of the declarations that are being called from the implicit call sites. There is a big difference between the test and non-test category. In the case of the both applications and libraries, most implicits used come from the standard library, followed by their external dependencies. The main sources of implicits in Scala are collections, concurrency and reflection packages together with the omnipresent `scala.Predef` object.

The collections are used by 80.3% projects (from 96.6% in large apps to 44.6% in tests). Most of the collection transforming operations such as `map`, use a builder factory passed as an implicit parameter `CanBuildFrom`. 38.3% of all implicit call sites involving methods that use this implicit parameter appear in libraries. Implicit parameters are used for reflection. Instances of `Manifest`, `ClassTag` or `TypeTag` classes can be requested from the compiler to be passed as implicit arguments, allowing one to

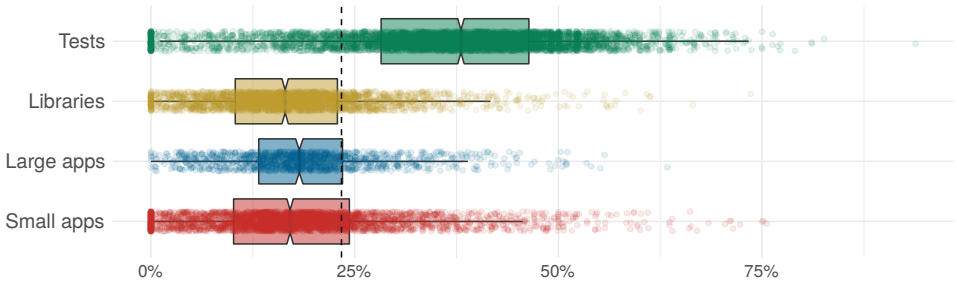


Fig. 7. Ratio of implicit call

get in-depth information about the type parameters of a method at runtime, circumventing the limitation of Java's type erasure. This is used a lot in large applications (90.2%). Less in libraries (61.7%) and small projects (56%) or tests (56.4%). Over half of all the large projects (58.5%) and third of libraries (32.1%) employ some concurrency routines from the Scala standard library. `scala.Predef` defines basic conversion like `String` to `StringOps` (extending the functionality of Java strings) or an arrow association, allowing one to use `a->b` to create a tuple of `(a,b)`. These are used by almost all the projects regardless of category.

Excluding the Scala standard library and testing frameworks, the rest of the implicits in the case of application and libraries come from a number of different external dependencies. There are some well known and projects with rich set of implicit usage such as the Lightbend/Typesafe stack with Play (a web-application framework, used in 5% of implicit call sites), Slick (object-relational mapping, 2.6%) or `akka` (an actor framework, 2.3%). These libraries define domain-specific languages which, in order to fit well in the host language yet to appear to introduce different syntactic forms, heavily rely on implicits. Next to a more flexible syntax (as compared to Java or C#), implicits are the main feature for embedding DSLs.

In the case of tests, the vast majority of implicits comes from project dependencies, which are dominated by one of the popular testing frameworks. These frameworks define DSLs in one form or another, striving to provide an API that reads like English sentences. For example a simple test:

```
"Monte Carlo method" should "estimate pi" in { MCarloPi(tries=100).estimate === 3.14 +- 0.01 }
```

contains six implicit call sites. Four are implicit conversions adding methods `should` to `String`, `in` to `ResultOfStringPassedToVerb` (the resulting type of calling the `should` method), `===` and `+-` to `Double`. Three of them additionally take implicit parameters for pretty-printing, source position (generated by a macro), test registration, and floating point operations. The implicit macro generating the source position is actually the single most used implicit parameter in the corpus with 912.7K

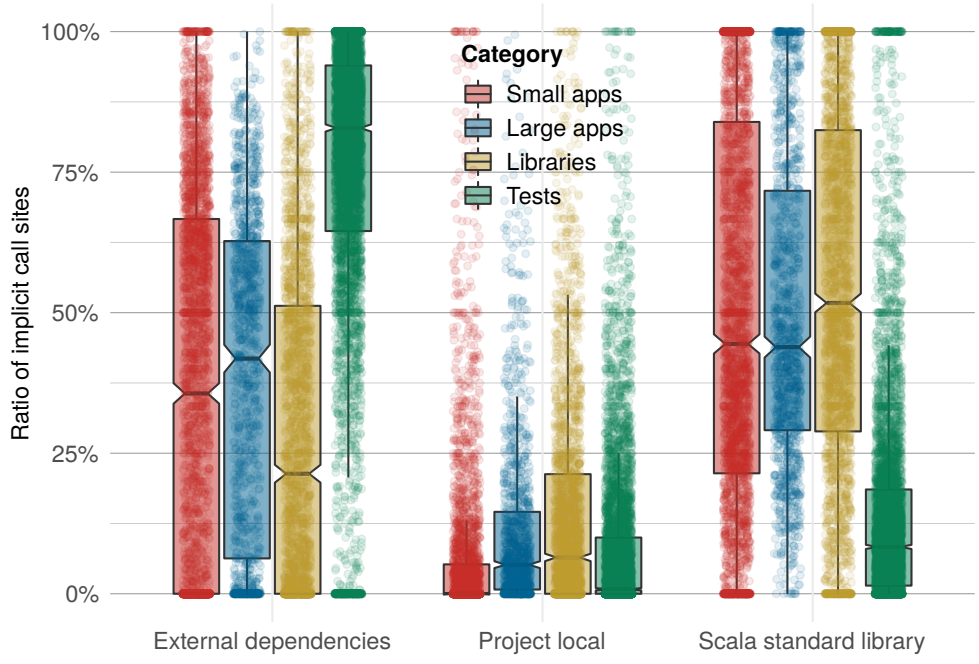


Fig. 8. Origin of parameter declarations

instances. Excluding the test frameworks, the ratio of implicit locations become very close to that of the main code, with collections and `scala.Predef` dominating the distribution.

5.1 Implicit Conversion

We recognize conversions by finding signatures that are either: (1) an `implicit def` with one non-implicit parameter (and 0+ implicit parameters) and a non-`Unit` return, or (2) an implicit `val`, `var` or `object` that extends a function type `T=>R` such that `R` is not `Unit`. Note, that `implicit class` declarations are already de-sugared into a class and a corresponding `implicit def`.

Table 4 summarizes conversions across the four categories of projects; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X across all categories and Z is a ratio of projects identified in the given category. As expected, the majority of implicit conversions (80%) are defined in libraries (52% of libraries define at least one conversion) while most use is in the tests (61% of all implicit conversion call sites).

Table 4. Conversions

	Small Apps	Large Apps	Libraries	Tests
Declarations	2K (04% 22%)	7K (13% 58%)	49K (80% 52%)	2K (03% 11%)
Call sites	89K (04% 88%)	384K (15% 99%)	514K (20% 94%)	1M (61% 95%)

Table 5 lists the projects declaring and using the most conversions; each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences. It is interesting to observe that the projects that define the most conversions are not necessarily the ones which use the most, as usage is likely correlated to project size.

Table 5. Top conversions

Project	Declarations	Project	Callsites
shadaj/slinky (265, 46K)	34K	exoego/aws-sdk-scalajs-facade (3, 302K)	130K
pbaun/rere (4, 14K)	446	scalatest/scalatest (782, 76K)	116K
etorreborre/specs2 (642, 26K)	440	apache/spark (21K, 238K)	60K
sisioh/aws4s (7, 15K)	402	akka/akka (10K, 109K)	30K
CommBank/grimlock (29, 22K)	385	gapt/gapt (48, 68K)	22K
scala/scala (11K, 139K)	346	ornicar/lila (5K, 175K)	17K
scalatest/scalatest (782, 76K)	343	psforever/PSF-LoginServer (28, 41K)	15K
scalan/special (2, 33K)	336	broadinstitute/cromwell (384, 65K)	15K
scalaz/scalaz (4K, 34K)	301	hmrc/tai-frontend (0, 31K)	14K
lift/framework (1K, 42K)	280	getquill/quill (1K, 11K)	14K

Conversions are used in 96.8% of all projects (7,050). There are 2.5M implicit conversions or 31.5% of all implicit call sites. This is understandable as it is hard to write code that does not, somehow, trigger one of the many conversions defined in the standard library. In fact, for application code 47.4% of implicit conversions have definitions originating in the standard library. Most conversions, 61.1% to be exact, happen in tests; for those, 59.4% of them have definitions that originate from one of the two popular testing frameworks (`scalatest` or `specs2`). If we exclude the standard library and testing frameworks, most conversions are defined in imported code, only about 18.8% are calls to conversions with definitions local to their project.

In terms of conversion declarations, 41.1% of projects (2,991) provide 61,995 conversions (16.7% of all declarations) with a median of 3 per project and a s.dev of 615.5. As expected, testing frameworks have many declarations (343 in `scalatest`, 440 in `specs2`). We note that `slinky` defines over 33.6K

conversions (almost all programatically generated). The reason is that this project aims at allowing one to writing React code (a JavaScript library for building user interfaces) in Scala in a similar manner as to that of JavaScript. This project is hardly used, we could find only 2 clients (with 3.6K LOC) that used 8 `slinky` conversions.

The most used conversion is `ArrowAssoc` as it enables users to create tuples with an arrow (*e.g.,* `1 -> 2`). The next most popular is `augmentString`, a conversion that allows users to use index sequence methods on `String` objects. On average, projects targeting JavaScript use 2.5 times more often implicit conversions than JVM projects. Most of these conversions come from libraries that simplify front-end web development with DSLs for recurring tasks such as DOM construction and navigation. Only 1.1K (0.3%) of the implicit conversions were defined with functional types (*i.e.,* using `implicit val`, `var` or `object`); this is good as implicit values that are also conversions can be the source of problems.

5.2 Implicit Parameters

We record all method and constructor declarations with implicit parameter list. Table 6 summarizes parameters across the four categories of projects; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X over all categories and Z is a ratio of projects in the given category.

Table 6. Parameters

	Small Apps	Large Apps	Libraries	Tests
Declarations	8K (06% 35%)	50K (32% 73%)	87K (55% 68%)	11K (07% 23%)
Call sites	134K (04% 89%)	749K (20% 99%)	691K (19% 94%)	2M (58% 95%)

Table 7 lists the projects declaring and using the most implicit parameters; each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences. As with conversion, the projects that define the most implicits are not necessarily the ones with most calls.

Table 7. Top implicit parameters

Project	Declarations	Project	Callsites
lampepfl/dotty (3K, 89K)	4K	scalatest/scalatest (782, 76K)	242K
scalaz/scalaz (4K, 34K)	4K	apache/spark (21K, 238K)	59K
typelevel/cats (3K, 24K)	3K	typelevel/cats (3K, 24K)	53K
robertofischer/hackerrank (0, 50K)	2K	CommBank/grimlock (29, 22K)	52K
scalatest/scalatest (782, 76K)	2K	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
sirthias/parboiled2 (604, 6K)	1K	akka/akka (10K, 109K)	43K
laserdisc-io/laserdisc (23, 7K)	1K	monix/monix (1K, 34K)	40K
slamdata/quasar (742, 27K)	1K	scalaz/scalaz (4K, 34K)	39K
etorreborre/specs2 (642, 26K)	984	slamdata/quasar (742, 27K)	31K
EHRI/ehri-frontend (10, 68K)	981	lampepfl/dotty (3K, 89K)	29K

Calls sites with implicit parameters are frequent, they account for 46.2% (3.7M) of all Scala call sites. As shown in Table 6, tests account for 58% of these calls. Small applications have a lower proportion, most likely because they account for relatively few lines of code.

In terms of declarations, 78.2% of projects (5.7K) have over 370.7K implicit parameter declarations. The remaining projects do not declare any. The majority, 89.6% (332.2K), of declarations are public. Over half of the declarations come from 200 projects which often implement DSL-like APIs. This also happens internally in applications. For example, `ornicar/lila`, an open source chess server, is one of the largest and most popular apps in the corpus. It uses implicits for a small database management DSL.

5.3 Idioms and Patterns

In this subsection, we look at popular implicit idioms and answer the question how frequently are these idioms used. For each, we describe the heuristic used to recognize the pattern and give a table with the 10 top most projects in terms of declarations as well as in use in terms of call sites. Each of the table has the same structure: each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences for declarations and call sites.

Table 8 gives a summary of the declaration and uses of the various idioms and patterns split by our code categories; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X over all categories and Z is a ratio of projects in the given category.

Table 8. Idioms and patterns

Pattern	Small Apps	Large Apps	Libraries	Tests
Late Trait Implementation	278 (08% 04%)	968 (28% 15%)	2.1K (59% 14%)	177 (05% 01%)
Extension Methods	1.7K (09% 17%)	5.1K (28% 48%)	10.5K (57% 45%)	1.2K (06% 08%)
Type Classess	4.3K (05% 19%)	17.2K (21% 49%)	54.2K (67% 53%)	5.8K (07% 15%)
Extension Syntax Methos	1.3K (06% 09%)	4.3K (20% 28%)	13.9K (66% 31%)	1.6K (08% 06%)
Type Proofs	110 (06% 01%)	320 (18% 05%)	1.3K (73% 06%)	39 (02% 00%)
Context	5K (06% 25%)	34.9K (41% 62%)	39.2K (46% 50%)	5.7K (07% 14%)
Unrelated Conversions	672 (02% 07%)	2.3K (06% 26%)	38.1K (92% 20%)	441 (01% 03%)
Bidirectional Conversion	197 (17% 01%)	321 (28% 06%)	556 (49% 03%)	61 (05% 00%)

(a) Declarations

Pattern	Small Apps	Large Apps	Libraries	Tests
Late Trait Implementation	21.4K (07% 54%)	67.3K (22% 84%)	97.8K (31% 54%)	125.4K (40% 47%)
Extension Methods	40.9K (03% 68%)	207.7K (13% 95%)	250.7K (15% 82%)	1.1M (69% 90%)
Type Classess	99.4K (05% 86%)	502.2K (23% 99%)	544K (25% 92%)	1.1M (48% 88%)
Extension Syntax Methos	42.7K (03% 55%)	213.5K (16% 89%)	227.5K (17% 61%)	881K (65% 75%)
Type Proofs	1.7K (03% 19%)	10.6K (19% 61%)	14.9K (27% 44%)	28.8K (51% 19%)
Context	35.9K (02% 60%)	239.2K (14% 87%)	154.6K (09% 61%)	1.3M (75% 84%)
Unrelated Conversions	29.7K (07% 72%)	107.4K (25% 96%)	112.9K (26% 78%)	178.1K (42% 57%)
Bidirectional Conversion	1.9K (06% 13%)	7.9K (25% 42%)	8.8K (28% 26%)	13.2K (41% 13%)

(b) Call sites

5.3.1 Late Trait Implementation. Late traits are recognized by looking for `implicit def T=>R` where `R` is a Scala trait or Java interface. Technically, the same effect can be achieved with an `implicit class` extending a trait, but in all cases the implicit class adds additional methods, and thus is disqualified. As Table 8 shows there are only a few declarations of this pattern, mostly in libraries. Table 9 gives the top 10 projects using late traits.

Most conversions, 79.8%, are used between types defined in the same project. Conditional implementation account for 16.4% of this pattern. 19.7% convert Java types (from 176 different libraries). Focusing on the JDK, 53 conversions are related to I/O, 50 are from Java primitives and 27 involve time and date types. There are 990 conversions from Scala primitives with `String` (217) and `Int` (77) being the most often converted from.

5.3.2 Extension Methods. In general extension methods can be defined using both `implicit class` and `implicit def`. While the former is preferred, the latter is still being used. Since an `implicit def` can be also used for late trait implementation or to simply relating two types, we only consider

Table 9. Top late traits

Project	Declarations	Project	Callsites
lift/framework (1K, 42K)	152	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
lampepfl/dotty (3K, 89K)	106	scalatest/scalatest (782, 76K)	9K
etorreborre/specs2 (642, 26K)	94	akka/akka (10K, 109K)	6K
scala/scala (11K, 139K)	82	CommBank/grimlock (29, 22K)	4K
CommBank/grimlock (29, 22K)	81	hmrc/tai (1, 13K)	3K
scalatest/scalatest (782, 76K)	74	broadinstitute/cromwell (384, 65K)	3K
l-space/l-space (3, 17K)	68	maif/izanami (91, 19K)	2K
anskarl/auxlib (1, 1K)	63	etorreborre/specs2 (642, 26K)	2K
anskarl/LoMRF (58, 13K)	63	mattpap/mathematica-parser (24, 476)	2K
squeryl/squeryl (521, 9K)	49	playframework/play-json (193, 5K)	2K

`implicit def` with a return type that is neither a Scala trait nor a Java interface and that is defined in the same file as the conversion target because extension methods are usually collocated in either the same compilation unit or in the source file. We found 12,150 implicit classes, 65.3% of all extension methods. Table 8 shows that extension methods are widely used, they are defined across the corpus and in particular in large applications and libraries. Their use is widespread as well. The top 10 projects using extension methods appear in Table 10.

Table 10. Top extension methods

Project	Declarations	Project	Callsites
pbaun/rere (4, 14K)	428	scalatest/scalatest (782, 76K)	87K
etorreborre/specs2 (642, 26K)	295	exoego/aws-sdk-scalajs-facade (3, 302K)	46K
scalaz/scalaz (4K, 34K)	281	apache/spark (21K, 238K)	24K
scalan/special (2, 33K)	248	akka/akka (10K, 109K)	22K
lampepfl/dotty (3K, 89K)	214	hmrc/tai-frontend (0, 31K)	14K
ritschwumm/scutil (6, 12K)	214	getquill/quill (1K, 11K)	13K
typelevel/cats (3K, 24K)	171	hmrc/tai (1, 13K)	13K
lift/framework (1K, 42K)	168	monix/monix (1K, 34K)	12K
broadinstitute/cromwell (384, 65K)	166	broadinstitute/cromwell (384, 65K)	10K
monsantoco/aws2scala (19, 10K)	134	hmrc/iht-frontend (1, 49K)	10K

There are 1.9K conditional extensions (10.2%). From these, 1.6K are related to type classes and 323 to contexts. 1.7K instances extends Java types (9.3%) across 676 libraries. Similarly to late traits, the Java I/O (224), date and time (200) and Java primitives (59) are the most often extended. Extension methods are also used to extends Scala primitives (3.7K), again `String` and `Int` being the most popular (1,169 and 452 respectively). This is understandable as these are the basic types for building embedded DSL.

5.3.3 Type Classes. We recognize type classes from their instances that are injected by a compiler as implicit arguments. What differentiate them from an implicit argument is the presence of type arguments linked to type parameters available in the parent context. This is what distinguishes a type class and a context. For example, the following do not match:

```
def f(x: Int)(implicit y: A[Int])     def f[T](x: T)(implicit y: T)
```

while the following do:

```
def f[T](x: T)(implicit y: A[T])      implicit class C[T](x: T)(implicit y: A[T])
```

We match implicit parameters with at least one type argument referencing a type parameter. Table 8 shows that type classes are the most widely declared pattern. Both libraries and large application

use it frequently. They are also the most frequent call sites. The top 10 projects using type classes are in Table 11.

Table 11. Top type classes

Project	Declarations	Project	Callsites
scalaz/scalaz (4K, 34K)	4K	scalatest/scalatest (782, 76K)	96K
typelevel/cats (3K, 24K)	3K	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
robertofischer/hackerrank (0, 50K)	2K	typelevel/cats (3K, 24K)	48K
sirthias/parboiled2 (604, 6K)	1K	apache/spark (21K, 238K)	46K
slamdata/quasar (742, 27K)	1K	CommBank/grimlock (29, 22K)	43K
laserdisc-io/laserdisc (23, 7K)	1K	scalaz/scalaz (4K, 34K)	38K
scalatest/scalatest (782, 76K)	947	slamdata/quasar (742, 27K)	30K
twitter/algebird (2K, 24K)	899	laserdisc-io/laserdisc (23, 7K)	18K
scalanlp/breeze (3K, 36K)	887	scalaprops/scalaprops (226, 6K)	17K
nrinaudo/kantan.csv (244, 5K)	832	nrinaudo/kantan.csv (244, 5K)	16K

Type classes are involved in 30% of the implicit calls which use over 11K type classes. Type classes are dominated by the standard library (42%). As expected, most come from the collection framework, `scala.Predef` and the `math` library. Next are testing libraries (15%) followed by the some of the most popular frameworks and libraries including `Typelevel cats` and `scalaz` that provide basic abstractions for functional programming, including a number of common type classes. These two libraries are used by almost 40% in the corpus.

5.3.4 Extension Syntax Methods. From extension methods we select instances that define implicit parameters that match out type class definition from Section 5.3.3. Summary is in Table 12.

Table 12. Top extension syntax methods

Project	Declarations	Project	Callsites
pbaun/rere (4, 14K)	428	scalatest/scalatest (782, 76K)	87K
etorreborre/specs2 (642, 26K)	295	exoego/aws-sdk-scalajs-facade (3, 302K)	46K
scalaz/scalaz (4K, 34K)	281	apache/spark (21K, 238K)	24K
scalan/special (2, 33K)	248	akka/akka (10K, 109K)	22K
lampepfl/dotty (3K, 89K)	214	hmrc/tai-frontend (0, 31K)	14K
ritschwumm/scutil (6, 12K)	214	getquill/quill (1K, 11K)	13K
typelevel/cats (3K, 24K)	171	hmrc/tai (1, 13K)	13K
lift/framework (1K, 42K)	168	monix/monix (1K, 34K)	12K
broadinstitute/cromwell (384, 65K)	166	broadinstitute/cromwell (384, 65K)	10K
monsantoco/aws2scala (19, 10K)	134	hmrc/iht-frontend (1, 49K)	10K

We found 18.6K of syntax methods instances in 2.5K projects. Most of them are defining operations of generic algebraic data types.

5.3.5 Type Proofs. We recognize this pattern by select `implicit def` that take generalized type constraints, such as equality (`=:=`), subset (`<:<`) and application (`=>`) as implicit type parameters. Summary is in Table 13.

This revealed a very few projects (270) besides Scala itself and related projects (the new Scala 3 compiler). They define 1.6K methods taking type proofs as implicit parameters. Most of them are small applications which seem to be projects experimenting with type level programming. There are however interesting use cases. Manually inspecting the bigger projects we found common use cases, both are related to enforcing certain API restrictions at compile time. In one case (`scalajs-react`—another project bringing React application development into Scala), it is used to ensure that a given

Table 13. Top type proofs

Project	Declarations	Project	Callsites
scalatest/scalatest (782, 76K)	167	CommBank/grimlock (29, 22K)	5K
scalikejdbc/scalikejdbc (982, 13K)	91	akka/akka (10K, 109K)	3K
scalanlp/breeze (3K, 36K)	67	typelevel/cats (3K, 24K)	2K
mpollmeier/gremlin-scala (412, 2K)	54	outworkers/phantom (1K, 12K)	2K
playframework/play-json (193, 5K)	45	scalatest/scalatest (782, 76K)	2K
xuwei-k/applybuilder (7, 767)	42	sisioh/aws4s (7, 15K)	2K
japgolly/test-state (108, 6K)	39	laserdisc-io/laserdisc (23, 7K)	1K
NICTA/scoobi (487, 13K)	34	apache/spark (21K, 238K)	623
scoundrel-tech/scoundrel (0, 10K)	34	tixxit/framian (118, 7K)	546
twitter/scalding (3K, 29K)	34	scoundrel-tech/scoundrel (0, 10K)	541

method is called only once. Another instance (`finagle`, an RPC system) creates a type-safe builder pattern that throws a compile-time error in the case the constructed object is missing required field. In both cases authors used `@implicitNotFound` annotation to provide customized error message.

5.3.6 Context. Whether or not an implicit argument is an instance of the context pattern is hard to quantify, since it depends on intent. We recognize them by selecting implicit call sites that are neither labeled as a type class application nor as a type proof. Summary is in Table 14.

Table 14. Top context

Project	Declarations	Project	Callsites
lampepfl/dotty (3K, 89K)	4K	scalatest/scalatest (782, 76K)	201K
scalatest/scalatest (782, 76K)	1K	apache/spark (21K, 238K)	38K
sirthias/parboiled2 (604, 6K)	1K	akka/akka (10K, 109K)	28K
EHRI/ehri-frontend (10, 68K)	779	monix/monix (1K, 34K)	27K
ornicar/lila (5K, 175K)	774	lampepfl/dotty (3K, 89K)	26K
ponkotuy/MyFleetGirls (86, 26K)	717	CommBank/grimlock (29, 22K)	18K
Sciss/SoundProcesses (23, 13K)	715	hmrc/iht-frontend (1, 49K)	18K
sciss/fscape-next (6, 27K)	696	hmrc/tai-frontend (0, 31K)	17K
ruimo/store (5, 38K)	688	gapt/gapt (48, 68K)	16K
sciss/patterns (1, 8K)	620	twitter/finagle (7K, 64K)	13K

As expected, contexts are used heavily in projects such as Scala compiler (`dotty` is the new Scala compiler), `spark` or `akka`, *i.e.,* projects that are centered around some main context which is being passed around in number of methods. Java types are also used as context parameters. Together 50 types from JDK are used in 1.8K methods across 179 projects. The top used one is `SQLConnection` followed by interfaces from `java.io`. Scala primitive types are used in 1,044 methods in 159. Function types are also used as contexts (645 methods in 154 projects), providing a convenient way to define application counters, implicit filters and other default data processors.

5.3.7 Anti-pattern: Conversions. Unrelated conversions are public, top-level definitions defined outside of either from or to compilation units. We recognize them by selecting implicit conversions that are not block-local, or private, or protected and are not defined in the same compilation unit as the source or the target type. Summary is in Table 15.

There are 41.5K of unrelated conversions spanning across 1.2K projects (16.2%). Most of them (33.6K) belong to the already mentioned `slinky` projects bringing React apps development to Scala. They are used in 6.1K (83.9%) projects. If we change the query to only regard the same artifact then it drops too 1.9K conversions in 619 projects. There are some indication that unrelated conversions

Table 15. Top unrelated conversions

Project	Declarations	Project	Callsites
shadaj/slinky (265, 46K)	34K	apache/spark (21K, 238K)	21K
sisioh/aws4s (7, 15K)	402	gapt/gapt (48, 68K)	11K
CommBank/grimlock (29, 22K)	299	scalatest/scalatest (782, 76K)	8K
scala/scala (11K, 139K)	166	akka/akka (10K, 109K)	7K
etorreborre/specs2 (642, 26K)	130	CommBank/grimlock (29, 22K)	6K
squeryl/squeryl (521, 9K)	128	ornicar/lila (5K, 175K)	6K
scoundrel-tech/scoundrel (0, 10K)	113	ilya-klyuchnikov/tapl-scala (126, 13K)	3K
scala/scala-java8-compat (353, 4K)	112	scoundrel-tech/scoundrel (0, 10K)	3K
typelevel/cats (3K, 24K)	110	broadinstitute/cromwell (384, 65K)	2K
lift/framework (1K, 42K)	101	mattpap/mathematica-parser (24, 476)	2K

might be deprecated in the upcoming revision of the Scala language[12]. The numbers here show, that these conversions are being defined, but they are usually in the scope of the same library. From the unrelated conversions, 1.6K from 552 projects involves Scala primitive types. They are present in all categories, but majority comes from libraries where they are used as building blocks for DSLs. Only a very few (81 in 47 projects) convert just between primitive types.

For the conversions that go both ways, we consider all pairs of such conversions that are defined in the same artifact and thus could be easily imported in the same scope.

We have identified 1.1K such conversions defined in 209 (2.9%) projects and used across 1.9K (26.5%) projects. As expected, this has matched all the Scala-Java collection conversions defined in the `scala.collection` package. They are used in 728 (10%) projects. This is significantly less than the recommended alternative using explicit `asJava` or `asScala` decorators that are being used by 22.4% of projects. 244 projects mix both approaches.

From a manual inspection of some of the other popular bi-directional conversion, we find that it is used in libraries that provide both Java and Scala API (*e.g.,* `spark` or `akka`) allowing one to freely mix Java and Scala version of the classes. Some libraries use them to provide easier syntax for its domain objects (*e.g.,* using a tuple to represent a cell coordinate), or lifting types from/to `scala.Option`. Another distinct category are conversions between many of the different date and time representations in both Java and Scala. We found only 129 bi-directional conversions involving primitive types, out of which 13 are only between primitives.

5.4 Complexity

One question we wanted to address was the amount of work performed by the Scala compiler. This is motivated by the need for the programmer to reverse engineer the compiler's work to understand how to fix their code when an error is related to implicits. In terms of code size, if one were to sum up the length of the symbols inserted by the compiler at the various call sites that use implicit arguments, this would amount to 55M characters or about 3.5x the size of the entire Scala project. Figure 9 shows the distribution of injected implicit arguments into methods. We limit the graph to 10 injected arguments, but in practice there is a long tail. The measurements are obtained by inspecting each call site where implicit resolution is involved and counting arguments injected directly to the target function as well as arguments injected to nested calls needed for the implicit derivation. While the distribution has a long tail, going all the way to 5,695, the median is 1. At the extreme, the `xdotai/typeless` project is exploring type-level programming and has one call site that includes 5,695 nested implicit calls and value injection. Expressed in length of the injected code, that call site has the compiler inject 56.2K characters. Figure 10 shows the distribution of

[12] *cf.* https://github.com/lampepfl/dotty/pull/2060

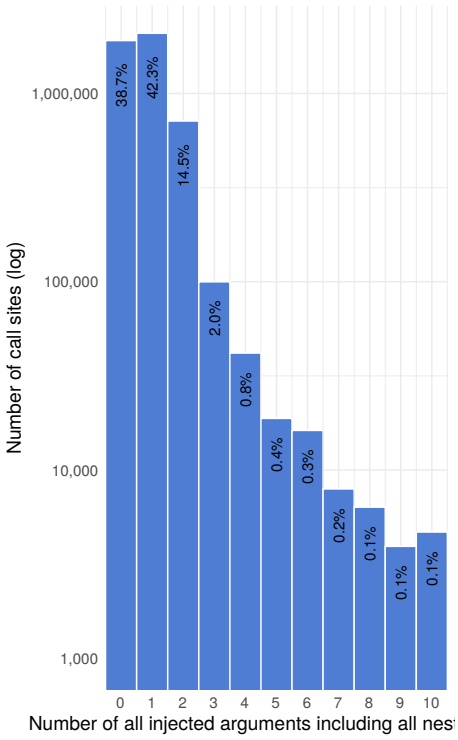


Fig. 9. Injected arguments

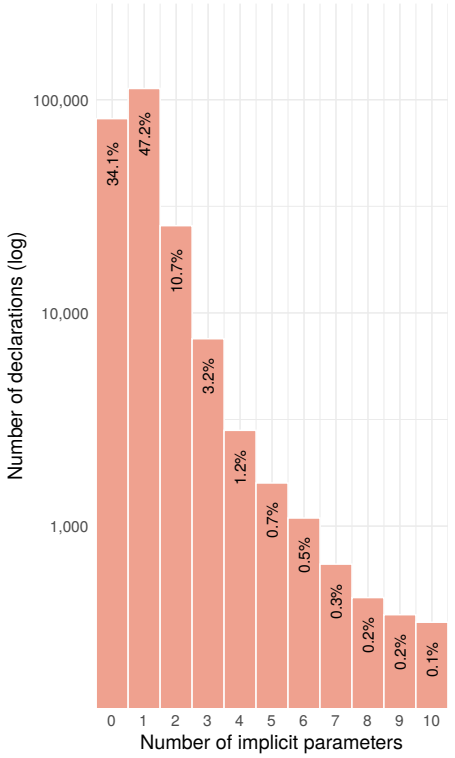


Fig. 10. Implicit parameters

the number of implicit parameter declarations. The data suggests that programmers are likely to encounter functions with one or two implicits rather frequently. And they are likely to deal with functions with four or more implicits several times per project.

To help navigate this complexity, the Scala plugin for Intellij IDEA has a feature that can show implicit hints, including implicit resolution in the code editor. This effectively reveals the injected code making it an indispensable tool for debugging. However, turning the implicit hints on severely hinders the editor performance creating a significant lag when working with implicits-heavy files. The second problem with this is that the Intellij Scala compiler is not the same as `scalac` and implicit resolution often disagrees between compiler implementations (*e.g.*, Intellij does not consider implicit shadowing in lexical context). Another way to mitigate some of the complexity related to errors occurring during resolution is to customize the error message emitted when an implicit type is not found. Scala provides the `@implicitNotFound(message)` annotation to this end, where `message` can be parameterized with the names of type parameters that the type defines. In the corpus, we have found it defined 1.2K times in 436 projects, and used in 110.9K call sites.

5.5 Overheads

Another question we are interested to investigate is the effect of implicits on compile time. We have demonstrated that on a synthetic example, resolution can significantly impact type-checking performance. There are 1,969 (8.4M LOC) using Scala 2.12.4+ for which we can get compile time statistics using the `-Ystatistics:typer` compiler flag. Furthermore 488 projects (2.8M LOC) use the `shapeless` library which is the most common approach to guide the type class derivation [Cantero 2018]. The result of measuring compilation speed between these two sets of projects is shown in Figure 11. More precisely, the figure shows data for projects that have more than 1,000 lines of

code (for smaller projects compilation times may be dominated by startup costs). The x-axis shows the density of implicit call sites (their ratio per line of code, ranging between 0 and almost 2). The y-axis shows compilation speed measured in lines per second. For this figure we capture the entire compilation time of each project, including I/O. Higher is better on this graph. Colors distinguish projects that use type classes (red) from those who do not (blue). The lines indicate an estimate of the conditional mean function (loess). If implicits were not influencing compilation time, one would expect both lines to be roughly flat and at the same level. What we see instead confirms our hypothesis, the cost of compilation increases with the density of implicits and the use of type classes further reduce compilation speed.

Another manifestation can be found in the `scalatest` testing framework. It defines a `Prettifier` for pretty printing which looks like a perfect candidate for a type class, yet the authors have decided to use it as a context parameter instead. The reason given for that is performance: "*Prettifier is not parameterized ... because assertions would then need to look up Prettifiers implicitly by type. This would slow compilation.*" In the corpus there are over 563.6K calls to methods using the `Prettifier` context. Resolving all of them implicitly using the implicit type class derivation machinery could indeed induce a slowdown across 2.5K projects.

5.6 Threats to Validity

We report on two source of threats to validity. One threat to *external validity* is linked to selection of code that was analyzed. We analyzed 15% of the Scala code publicly-available on GitHub. Our findings only generalize to industry if the code we analyzed is representative of industrial use of implicits. It is possible, for instance, that some companies enforce coding guidelines that impact the usage of implicits. We have no evidence that this is the case, but cannot rule it out. In terms

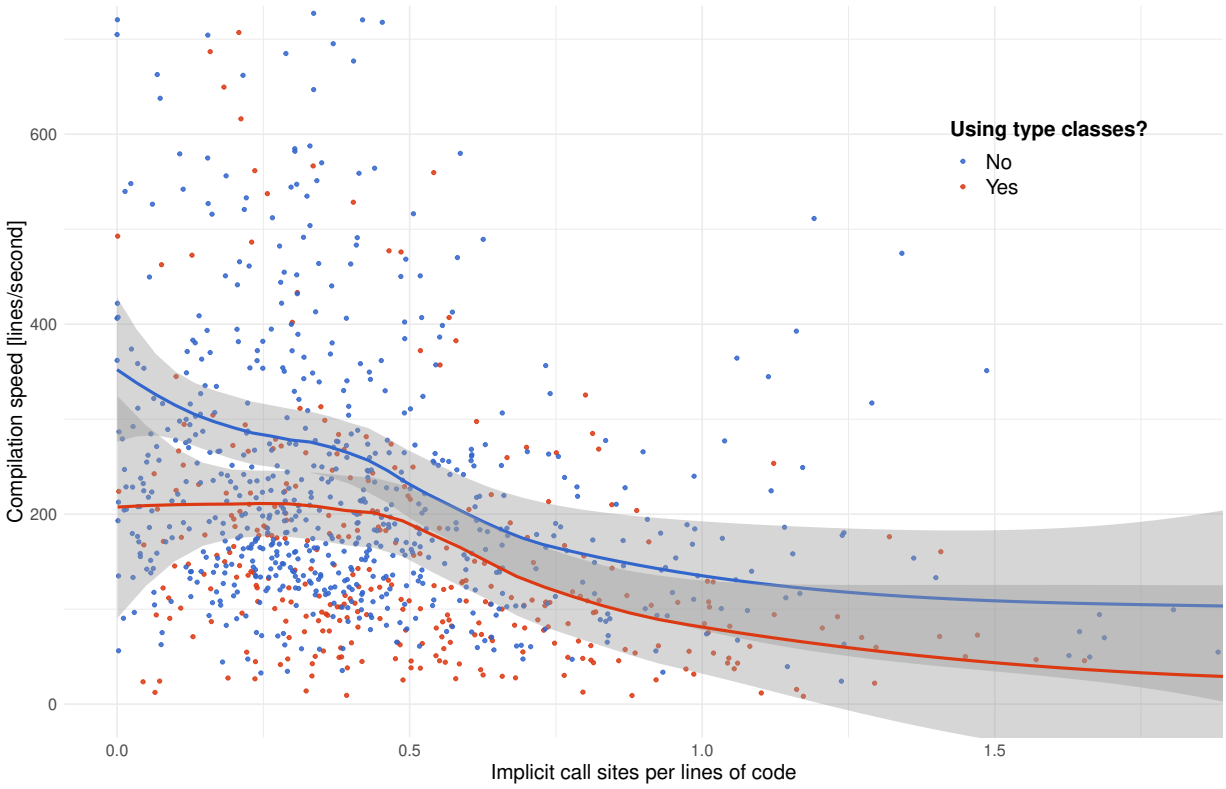


Fig. 11. Compilation slowdown

of threats to *internal validity* we consider our data analysis pipeline. It has several sources of inaccuracies. We rely on Scalameta to gather synthetic call sites. Scalameta restricts us to two Scala versions and it only generates metadata for about half of the selected projects. We are also aware that for 3% of implicit uses symbols could not be resolved.

6 RELATED WORK

The design of implicits as it appears in Scala is but one point in a larger space. While alternative designs are out of the scope of this work, we mention some important related work. Oliveira et al. [2010] established the connection between Haskell's type classes and Scala implicits with multiple examples. Oliveira et al. [2012] formalized the key ideas of implicits in a core calculus. Rouvoet [2016] expanded the Oliveira et al. work and proved soundness and partial completeness independent of termination. Schrijvers et al. [2019] present an improved variant of the implicit calculus. One key property of this work is the notion of *coherence* (which is attributed to Reynolds [1991]). Coherence requires a program to have a single meaning, i.e. it precludes any semantic ambiguity. Scala eschews coherence in favor of expressivity by allowing overlapping implicits. Schrijvers et al. propose a design that recovers coherence.

There have been efforts to study how Scala is used by practitioners. Tasharofi et al. [2013] looked at how often and why Scala developers mix the actor model with other models of concurrency. They analyzed only GitHub 16 projects at the compiled byte-code level with a custom tool. The choice of byte-code had some drawbacks. For example, their analysis could not detect indirect method invocations and thus they had to supplemented it with manual inspection. The same corpus is used by Koster [2015] to analyze different synchronization mechanisms used in Scala code. Despite using the same projects, he analyzed 80% more lines of code as the projects were updated to their latest commit. The increase was mostly due to `spark` that grew from 12K to 104K lines of code. Unlike the previous study, he opted for source code analysis based on string matching. De Bleser et al. [2019] analyzed the tests of 164 Scala projects (1.7M LOC) for a diffusion of test smells. They used a similar way of assembling a corpus. While they started with 72K projects, but only managed to compile 2.9K projects. They discard projects with less than 1K LOC or without `scalatest` unit tests. For analysis, they also used semantic data from the SemanticDB.

Pradel and Sen [2015] analyzed the use of implicit type conversions in JavaScript. They use dynamic analysis running hundreds of programs including the common JavaScript benchmarks and popular real-world websites. In JavaScript, implicit type conversion is basically a type coercion. Despite that the coercion rules are well formalized, they are fairly complex and confuse even seasoned JavaScript developers. Unlike in Scala that has static type system, JavaScript uses implicit type conversion extensively (it is present in over 80% of the studied programs), yet the study finds that over 98% of the conversion is what the authors consider as harmless.

7 CONCLUSIONS

Implicits are a cornerstone of the Scala programming language. There is hardly any API without them as they enable elegant architectural design. They allow one to remove a lot of boilerplate by leveraging the compiler's knowledge about the code. However, they can be also easily misused and if taken too far seriously hurt the readability of a code. Implicits are driven by type declarations. Thus, while, implicits are *used* transparently, with no indication in the program text, their application is guided by clear and precise rules. Our data shows that programmers have embraced them, with 98.2% of the projects we analyzed using them, and 78.2% of projects defining at least one implicit declaration. We also observed the prevalence of the idioms described, as most projects use them in some form. For implicit conversions, our results indicate that 96.8% of projects make use of them at some point, with the most popular conversions coming from the standard library and testing

libraries. From the idioms we presented in this paper, type classes and extension methods are used extensively. Regarding conversions, most convert to and from types within the scope of the project. However, there is a number of conversions defined on unrelated types. While deprecating this form of conversion has been discussed, doing so would break 1.2K projects (16.2%) in our corpus.

Observations for language designers. We have seen many source of complexities related to the notion of coherence. Future designs of implicits should strongly consider adopting some limits to expressivity in order to improve code comprehension. A related point is to avoid relying on names of implicits during their resolution as this leads to subtle errors. Better tool support and static analysis could help diagnose performance problems and could help code comprehension, but it is crucial that IDEs and the Scala compiler agree on how resolution is to be performed.

Observations for library designers. Over-engineered libraries are hard to understand. It is worth considering the costs and benefits of adding, for example, type classes to an API. Asking questions such as "Is retroactive extension an important use case?" or "How much boilerplate can actually be avoided?" may help target the right use-cases for implicits. Often the key design issue is whether good defaults can be provided. When they cannot, the benefits of implicits decrease significantly. A good library design is one that lets regular users benefit without forcing them fully understand the cleverness that the library designer employed. Finally, we leave designers with the following unsolicited advice: Do not use unrelated implicits! Do not use conversions that go both ways! Do not use conversions that might change semantics!

ACKNOWLEDGMENTS

Borja Lorente Escobar implemented an early version of the pipeline presented in this paper, we thank him for his contributions. We thank the reviewers for constructive comments that helped us improve the presentation. We thank Ólafur Páll Geirsson for his help with SemanticDB and Scalameta. We thank the members of the PRL lab in Boston and Prague for additional comments and encouragements. This work received funding from the Office of Naval Research (ONR) award 503353, from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 695412), the NSF (awards 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).

REFERENCES

Eugene Burmako. 2017. Unification of Compile-Time and Runtime Metaprogramming in Scala. (2017). https://doi.org/10.5075/epfl-thesis-7159

Jorge Vicente Cantero. 2018. Speeding Up Compilation Time with scalac-profiling. https://bit.ly/32gwTwP

Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing Diffusion and Perception of Test Smells in Scala Projects. In *International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2019.00072

Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/2487085.2487132

Li Haoyi. 2016. Implicit Design Patterns in Scala. https://web.archive.org/web/20180326160306/http://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html.

Joeri De Koster. 2015. *Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems.* Ph.D. Dissertation. Vrije Universiteit Brussel, Belgium.

R Lämmel and K Ostermann. 2006. Software extension and integration with type classes. *Conference on Generative Programming and Component Engineering (GPCE)*. https://doi.org/10.1145/1173706.1173732

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/325694.325708

Lightbend. 2018. Scala Developer Suvey. https://bit.ly/2Uk56sB.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA. https://doi.org/10.1145/3133908

Chris Marshall. 2009. Is the Scala 2.8 collections library a case of the longest suicide note in history? https://stackoverflow.com/questions/1722726/is-the-scala-2-8-collections-library-a-case-of-the-longest-suicide-note-in-hist.

Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. 2013. Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *Cconference on Object Oriented Pogramming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2509136.2509547

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-662-44202-9_13

Martin Odersky. 2017. What to leave implicit. https://www.youtube.com/watch?v=Oij5V7LQJsA. In *ScalaDays Chicago*.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: foundations and applications of implicit function types. *PACMPL* 2, POPL. https://doi.org/10.1145/3158130

Martin Odersky and Adriaan Moors. 2009. Fighting bit rot with types (experience report: Scala collections). In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*. https://doi.org/10.4230/LIPIcs.FSTTCS.2009.2338

Bruno Oliveira C. d. S., Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1869459.1869489

Bruno Oliveira C. d. S., Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The implicit calculus: a new foundation for generic programming. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2254064.2254070

Andrew Phillips and Nermin Serifovic. 2014. *Scala Puzzlers*. Artima Inc.

Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.519

John C. Reynolds. 1991. The coherence of languages with intersection types. *Theoretical Aspects of Computer Software* https://doi.org/10.1007/3-540-54415-1_70

Arjen Rouvoet. 2016. *Programs for Free: Towards the Formalization of Implicit Resolution in Scala*. Master's thesis. TU Delft.

Miles Sabin. 2019. Shapeless. https://github.com/milessabin/shapeless.

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. Cochis: Stable and Coherent Implicits. *Journal of Functional Programming* https://doi.org/10.1017/s0956796818000242

Joshua D Suereth. 2013. Implicit Classes. https://web.archive.org/web/20170922191333/https://docs.scala-lang.org/overviews/core/implicit-classes.html.

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

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-39038-8_13

Eric Torreborre. 2017. Achieving 3.2x Faster Scala Compile Time. https://jobs.zalando.com/tech/blog/achieving-3.2x-faster-scala-compile-time/

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/75277.75283

Bibliography

Adamsen, C. Q., Møller, A., Alimadadi, S., and Tip, F. (2018). Practical AJAX race detection for javascript web applications. In *Proc. 26th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

An, J.-h. D., Chaudhuri, A., Foster, J. S., and Hicks, M. (2011). Dynamic inference of static types for ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Andreasen, E., Gordon, C. S., Chandra, S., Sridharan, M., Tip, F., and Sen, K. (2016). Trace typing: An approach for evaluating retrofitted type systems. In *European Conference on Object-Oriented Programming (ECOOP)*.

Ball, T. (1999). The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, page 216–234, Berlin, Heidelberg. Springer-Verlag.

Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., Ernst, M. D., et al. (2015). Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679. IEEE.

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth and Brooks/Cole Advanced Books & Software, USA.

Boyapati, C., Khurshid, S., and Marinov, D. (2002). Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593.

Cantero, J. V. (2018). Speeding up compilation time with scalac-profiling. Online; accessed 1 December 2020. Link: https://web.archive.org/web/20201130093541/https://scala-lang.org/blog/2018/06/04/scalac-profiling.html.

Darcy, J. D. (2008). Kinds of compatibility: Source, binary, and behavioral. Online; accessed 1 December 2020. Link: https://web.archive.org/web/20201026024930/https://blogs.oracle.com/darcy/kinds-of-compatibility:-source,-binary,-and-behavioral.

De Bleser, J., Di Nucci, D., and De Roover, C. (2019). Assessing diffusion and perception of test smells in scala projects. In *International Conference on Mining Software Repositories (MSR)*.

Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. (2014). Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 779–790.

Elbaum, S. G., Chin, H. N., Dwyer, M. B., and Dokulil, J. (2006). Carving differential unit test cases from system test cases. In *International Symposium on Foundations of Software Engineering (FSE)*.

Ernst, M., Zhang, S., Saff, D., and Bu, Y. (2011). Combined Static and Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27.

Flückiger, O., Chari, G., Yee, M.-H., Ječmen, J., Hain, J., and Vitek, J. (2020). Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA). Link: https://doi.org/10.1145/3428288.

Furr, M., An, J.-h. D., and Foster, J. S. (2009). Profile-guided static typing for dynamic scripting languages. In *OOPSLA*.

Goel, A., Krikava, F., and Vitek, J. (2019). Rdt: A dynamic tracing framework for r. Link: https://doi.org/10.5281/zenodo.3625397.

Goel, A. and Vitek, J. (2019). On the design, implementation, and use of laziness in r. *Proc. ACM Program. Lang.*, 3(OOPSLA). Link: https://doi.org/10.1145/3360579.

Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR)*.

Haoyi, L. (2016). Implicit design patterns in scala. Online; accessed 1 December 2020. Link: https://web.archive.org/web/20180326160306/http://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html.

Huang, S., Guo, J., Li, S., Li, X., Qi, Y., Chow, K., and Huang, J. (2019). Safecheck: safety enhancement of java unsafe api. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 889–899. IEEE.

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

Jaygarl, H., Kim, S., Xie, T., and Chang, C. (2010). OCAT: object capture-based automated testing. In *nternational Symposium on Software Testing and Analysis (ISSTA)*.

Jensen, S. H., Jonsson, P. A., and Møller, A. (2012). Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis (ISSTA)*.

Jensen, S. H., Madsen, M., and Møller, A. (2011). Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Joshi, S. and Orso, A. (2007). SCARPE: A technique and tool for selective capture and replay of program executions. In *International Conference on Software Maintenance (ICSM)*.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101.

Koster, J. D. (2015). *Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems*. PhD thesis, Vrije Universiteit Brussel, Belgium.

Krikava, F. (2018). Automated unit test generation using genthat. In *User! conference*. Link: https://www.youtube.com/watch?v=qrX8q6euQII.

Krikava, F., Miller, H., and Vitek, J. (2019). Scala implicits are everywhere (artifact). Link: https://doi.org/10.5281/zenodo.3369436.

Křikava, F., Miller, H., and Vitek, J. (2019). Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA). Link: https://doi.org/10.1145/3360589.

Křikava, F. and Vitek, J. (2018a). Tests from traces: Automated unit test extraction for r. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 232–241, New York, NY, USA. Association for Computing Machinery. Link: https://doi.org/10.1145/3213846.3213863.

Křikava, F. and Vitek, J. (2018b). Tests from traces: Automated unit test extraction for r: Issta'18 artifact release. Link: https://doi.org/10.5281/zenodo.1306437.

Landman, D., Serebrenik, A., and Vinju, J. (2017). Challenges for static analysis of java reflection - literature review and empirical study. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518.

Legunsen, O., Hassan, W. U., Xu, X., Rosu, G., and Marinov, D. (2016). How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–613. IEEE.

Lewis, J. R., Launchbury, J., Meijer, E., and Shields, M. B. (2000). Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages (POPL)*.

Ligges, U. (2017). 20 Years of CRAN (video on channel9). In *Keynote at UseR!* Link: https://channel9.msdn.com/Events/useR-international-R-User-conferences/useR-International-R-User-2017-Conference/KEYNOTE-20-years-of-CRAN.

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA). Link: https://doi.org/10.1145/3133908.

Lopes, C. V. and Ossher, J. (2015). How scale affects structure in java programs. *ACM SIGPLAN Notices*, 50(10):675–694.

Madsen, M. and Lhoták, O. (2018). Implicit parameters for logic programming. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, pages 14:1–14:14, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/3236950.3236953.

Marshall, C. (2009). Is the scala 2.8 collections library a case of the longest suicide note in history? Link: https://web.archive.org/web/20210101150551/https://stackoverflow.com/questions/1722726/is-the-scala-2-8-collections-library-a-case-of-the-longest-suicide-note-in-hist.

Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., and Nystrom, N. (2015). Use at your own risk: the java unsafe api in the wild. *ACM Sigplan Notices*, 50(10):695–710.

Meawad, F., Richards, G., Morandat, F., and Vitek, J. (2012). Eval begone! semi-automated removal of eval from javascript programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 607–620.

Mezzetti, G., Møller, A., and Torp, M. T. (2018). Type regression testing to detect breaking changes in Node.js libraries. In *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*.

Miller, H., Haller, P., Burmako, E., and Odersky, M. (2013). Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Cconference on Object Oriented Pogramming Systems Languages and Applications (OOPSLA)*.

Miller, H., Haller, P., and Odersky, M. (2014). Spores: A Type-Based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming (ECOOP)*.

Møller, A. and Schwartzbach, M. I. (2018). Static program analysis. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

Møller, A. and Torp, M. T. (2019). Model-based testing of breaking changes in Node.js libraries. In *Proc. 27th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Morandat, F., Hill, B., Osvald, L., and Vitek, J. (2012). Evaluating the design of the r language: Objects and functions for data analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, page 104–131, Berlin, Heidelberg. Springer-Verlag. Link: https://doi.org/10.1007/978-3-642-31057-7_6.

Nakamaru, T., Matsunaga, T., Yamazaki, T., Akiyama, S., and Chiba, S. (2020). An empirical study of method chaining in java. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 93–102.

Nielsen, B. B., Torp, M. T., and Møller, A. (2020). Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(OOPSLA):187:1–187:25.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University.

Odersky, M. (2017). What to leave implicit, scaladays chicago keynote. Online; accessed 1 December 2020. Link: https://www.youtube.com/watch?v=0ij5V7LQJsA.

Odersky, M., Altherr, P., Cremet, V., Dragos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Spoon, L., Stenman, E., and Zenger, M. (2006). An overview of the scala programming language (2. edition).

Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., and Stucki, S. (2017). Simplicitly: foundations and applications of implicit function types. *PACMPL*, 2(POPL).

Odersky, M. and Moors, A. (2009). Fighting bit rot with types (experience report: Scala collections). In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*.

Oliveira C. d. S., B., Moors, A., and Odersky, M. (2010). Type classes as objects and implicits. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Raychev, V., Vechev, M., and Krause, A. (2015). Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124.

Richards, G., Gal, A., Eich, B., and Vitek, J. (2011a). Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

Richards, G., Hammer, C., Burg, B., and Vitek, J. (2011b). The eval that men do. In *European Conference on Object-Oriented Programming*, pages 52–78. Springer.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12.

Rooney, T. (2015). Automated test generation through runtime execution trace analysis. Master's thesis, Imparial College London.

Saff, D., Artzi, S., Perkins, J. H., and Ernst, M. D. (2005). Automatic test factoring for Java. In *International Conference on Automated Software Engineering (ASE)*.

Sen, K., Marinov, D., and Agha, G. (2005). Cute: A concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

Sozeau, M. and Oury, N. (2008). First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer.

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

Tasharofi, S., Dinges, P., and Johnson, R. E. (2013). Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming (ECOOP)*.

Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., and Schulte, W. (2009). Mseqgen: Object-oriented unit-test generation via mining source code. In *European Software Engineering Conference and Symposium on The Foundations of Software Engineering (ESEC/FSE)*.

Torreborre, E. (2017). Achieving 3.2x faster scala compile time. Online; accessed 1 December 2020. Link: https://web.archive.org/web/20201111223618/https://engineering.zalando.com/posts/2017/04/achieving-3.2x-faster-scala-compile-time.html.

Turcotte, A., Goel, A., Krikava, F., and Vitek, J. (2020a). Designing types for r, empirically (arifact). Link: https://doi.org/10.5281/zenodo.4037278.

Turcotte, A., Goel, A., Krikava, F., and Vitek, J. (2020b). Designing types for r, empirically (dataset). Link: https://doi.org/10.5281/zenodo.4091818.

Turcotte, A., Goel, A., Křikava, F., and Vitek, J. (2020c). Designing types for r, empirically. *Proc. ACM Program. Lang.*, 4(OOPSLA). Link: https://doi.org/10.1145/3428249.

Turon, A. (2017). Rust's language ergonomics initiative. Online; accessed 1 December 2020. Link: https://web.archive.org/web/20201112025600/https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html.

Urma, R.-G. (2017). Programming language evolution. Technical Report UCAM-CL-TR-902, University of Cambridge, Computer Laboratory. Link: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-902.pdf.

Villazón, A., Sun, H., Rosà, A., Rosales, E., Bonetta, D., Defilippis, I., Oporto, S., and Binder, W. (2019). Automated large-scale multi-language dynamic program analysis in the wild (tool insights paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad-hoc. In *Symposium on Principles of Programming Languages (POPL)*.

White, L., Bour, F., and Yallop, J. (2015). Modular implicits. *Electronic Proceedings in Theoretical Computer Science*, 198:22–63. Link: http://dx.doi.org/10.4204/EPTCS.198.2.