Comparing R Bytecode Compilers Written in R, Java, and Rust

Pierre Donat-Bouillud
Czech Technical University in Prague, Czech Republic
Northeastern University, USA

Filip Křikava
Czech Technical University in Prague, Czech Republic

Jakob Hain
Purdue University, USA

Adam Plodek
Vyper, Prague, Czech Republic

Jan Vitek
Czech Technical University in Prague, Czech Republic
Charles University, Prague, Czech Republic

Abstract

This paper presents a comparative analysis of three implementations of the R bytecode compiler: the official R implementation, a Java-based compiler, and a Rust-based compiler. The R compiler, written in R itself, poses challenges in terms of performance and maintainability. We evaluate designs of the compilers, their trade-offs, and performance characteristics. The Rust version outperforms the Java version, which itself outperforms the R version.

2012 ACM Subject Classification Software and its engineering → Dynamic compilers

Keywords and phrases R, bytecode, compiler

Digital Object Identifier 10.4230/OASIcs.Programming.2025.1

Category Extended Abstract

Supplementary Material *Software*: `https://github.com/PRL-PRG/rust-r-bcc` [4]
archived at `swh:1:dir:0dacc419ced335c4daf1d275cb34c26580acdf23`
Software: `https://github.com/PRL-PRG/r-compile-server/tree/main/server/src/main/java/org/prlprg/bc` [3]
archived at `swh:1:dir:80fa93336159947c757722b0eea284d792b20055`

Funding The research was supported by Czech Science Foundation Grant No. 23-07580X.

1 Introduction

R is a dynamic language mainly used by statisticians, which targets numerical computations more than tree traversals. In R, integers, real numbers or strings are actually vectors, *e.g.* `1` is an integer vector of size 1.

R is implemented in C and interoperates well with it, serving as glue for performance sensitive functions in native code. To boost actual R code performance, among other R optimization projects [9, 8], based on a JIT [11, 12, 5], the R interpreter added a bytecode interpreter [13] to its AST interpreter. The bytecode compiler is itself written in R, not in C, which presents unique challenges in terms of performance and expressivity.

As part of our ongoing effort to build a compilation service for R, we have implemented the R bytecode compiler in both Java and Rust. The goal is to output the very same bytecode as the original R compiler (byte-to-byte compatible), as a first step before adding optimizations, to make sure the new compilers are correct. However, unlike the R compiler, the alternative versions are separated from the rest of the R VM.[1] We compare the designs of the 3 compilers, and show performance numbers. Unsurprisingly, the Java and Rust versions win over the R implementation.

2 The R bytecode

When running R code, the source code is parsed into an AST represented as a R value, a SEXP. Then, the compiler may transform a SEXP into a bytecode object. The R bytecode is then executed by a stack-based VM entirely written in C.

Bytecode objects are bulky: they contain a bytecode *instructions* and a *constant pool*. Bytecode *instructions* are represented by a vector of integers. The *constant pool* stores the constant values in the bytecode instructions, the AST of the compiled expression and of subexpressions, and source locations for each instruction. There are over 129 fat instructions and some of them perform a lot of work including calling back to the AST interpreter.

The compiler is single pass and does very few optimizations: constant folding and inlining. If a call cannot be inlined, it emits it as a call to a closure, which might or might not be bytecode compiled.

Constant folding. It is applied directly on the AST, not on the generated bytecode. Only some types (*e.g.* numbers, booleans and strings) and when they are less than a predefined maximal size, and only some predefined functions, *e.g.* math operators, are folded.

Inlining. It is crucial because most syntax in R, even `if` and `{`, is a function. Without inlining, most code is actually passed to the AST interpreter to be run. As the user can modify any functions, the compiler inserts a guard that checks wether the target function has been masked at runtime. Only a fixed set of functions, including control flow, math, stats, assignments and vector operations, can be inlined.

Function bodies can be explictly compiled with `cmpfun` or `cmpfile`. Functions that are "large enough" are also JIT-compiled after 1 or 2 invocations. Upon installation of a package, all its functions are compiled.

3 Implementations

As we want to get the same bytecode as the R compiler, the Rust and Java 22 versions roughly follow the same single pass design, but use different language features and data structures.

3.1 R

The R bytecode compiler [13] is an R package, named `compiler`. It is written in the literate programming style with `noweb` [7]. Out of the 7000 nowweb lines, about 3000 are R code and the rest is LaTeX. There are a few parts written in C located outside the package in R's source code within `eval.c`. The R bytecode compiler, as a package, is also compiled to bytecode itself.

[1] The R VM connects to a compile service through gRPC (`https://grpc.io/`) which also caches the compiled bytecode.

The compilation is architectured around a *code buffer*, which is a list containing an instruction stream buffer, a constant pool, and closures to write the bytecode instructions and in the constant pool. Both instruction buffers and constant pools are implemented as R generic vectors, *i.e.,* an array of pointers. R has copy-on-write semantics and the search for a constant in the constant pool is performed with a linear search written in C.

3.2 Java

The Java implementation makes use of recent Java features, such as `record` or pattern matching. We use pattern matching on the SEXPs, but Java's support of it is limited.

The instruction stream uses the builder pattern and stores the instructions in an `ArrayList`. Contrary to the R version, bytecode instructions are typed. We use Java annotations to decorate the instruction definitions with their number of pops and pushes (`StackEffect`), and names of labels for pretty-printing control flow instructions (`LabelName`).

Due to the typed instructions, the constant pool is indexed with a *typed index*. To deduplicate indexes, it uses a `Map` instead of a linear search, and values are stored in an `ArrayList`. Deserializing the R bytecode into the typed form entails an overhead as all jump targets must be recomputed.

3.3 Rust

The Rust compiler stores allocations in a bump-allocated arena,[2] so lifetimes are only a small hindrance even with cyclic data-structures. A previous version of the Rust compiler heavily used `Box` and `clone`, but it was verbose, annoying to write, and slower than the java version. The bytecode is represented as a simple `enum` and they are stored in a `Vec` of bytes.

4 Correctness

To check the correctness of the bytecode compilers, we compare the outputs of the Rust and Java versions to the R version. We use *snapshot testing*: the R interpreter is run to generate the bytecode for crafted R functions, and functions from `base`, `utils`, `graphics`, `methods`, and `stats` packages. For the Java version, we developed our own snapshot testing framework, and for Rust, we use `insta`.[3]

5 Code size

The R version has 2 919 lines of codes, where the Java version has 8 811 lines and the Rust one, 6 690, as shown on Fig 1. However, the Java and Rust versions required to write the SEXP and serialization handling, which already exists in the R version.

6 Performance

We compile all functions in `base`, `utils`, `compiler`, `tools`, and `stats` packages, in total 97K lines, and measure the time to compile, excluding deserialization, repeating the measurement 10 times. Some packages have functions that compile to many opcodes (`utils`, `tools`, `stats`) while others to fewer opcodes, *i.e.* `compiler`, the R bytecode compiler itself, or `base`, which

[2] `https://docs.rs/bumpalo/latest/bumpalo/`
[3] `https://insta.rs/`

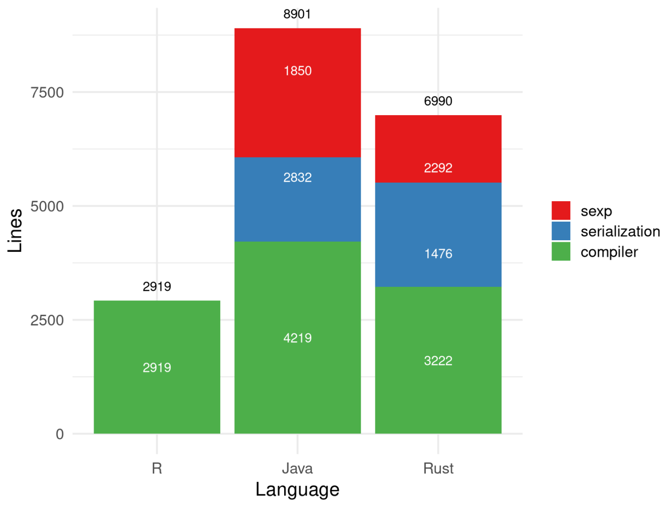

Figure 1 Number of lines of code for the R, Java, and Rust bytecode compilers, excluding comments and new lines (computed using `cloc` [2]).

for many functions just performs a call to a C binding (Table 1). For the Java version, for each of those 10 iterations, we compile all the functions a first time and then repeat to warmup the JIT.

Table 1 Number of functions, cumulated size of the bytecode in opcodes, and average number of opcodes per function, for each package in the microbenchmark.

	`base`	`utils`	`compiler`	`tools`	`stats`
#functions	1 144	512	139	774	925
#opcodes	125 206	117 161	15 185	282 765	199 615
Avg #opcodes/function	109	228	109	365	215

On an Apple M4 Pro with 12 cores and 24 GB of RAM, the R version takes about 17 seconds to compile all the functions. The Java version is around 33x quicker than the R version, and the Rust version is around 48x quicker than the R version, as seen on Fig. 2.

We also measure the peak heap memory usage of each of the compilers. For R, we call the `gc()` and for Java, we use the `Runtime` class, which give information about the memory usage with their respective GCs. For Rust, we use `valgrind`[10] with the `massif` heap profiler. Rust has the highest peak heap memory usage, with 1GB, as the arenas are not freed until the end. The Java version has a peak memory usage of 200MB in average, and the R version has a peak memory usage of 300MB.

7 Takeaways

Ergonomics. The R implementation is rather complicated to read, maintain and extend, and would be even more so without the literate programming style. For example, one of the features, complex assignments, requires a 3 page description just to introduce the code in the generated pdf documentation. In Rust and Java, we have a clear interface for the compiler and clear dependencies, unlike the R compiler which is intertwined with the rest of the VM and for instance sometimes call the AST interpreter.

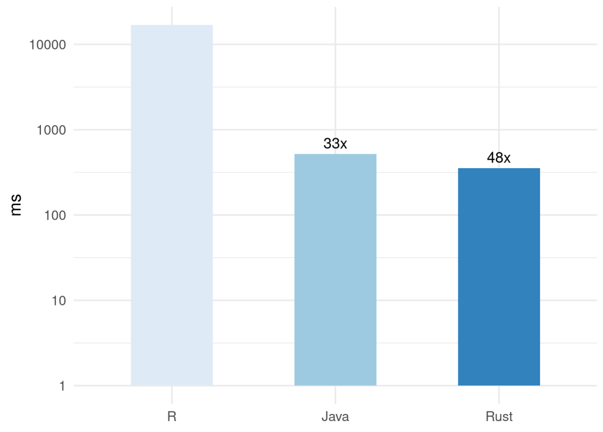

Figure 2 Compilation time for the packages in the microbenchmark depending on the language. The y-axis is logarithmic. We indicate the speedup compared to R on top of the Java and Rust bars.

Another pain point of R is its lack of complex data structures and checked type annotations. For instance, there are no real hash maps [4] in R, so R uses a linear search to find duplicates in the constant pool. On the contrary, the Java and Rust codes are more readable and expressive thanks to Java's pattern matching and algebraic datatypes. We think that going away from R for the compiler would make it possible to write more complex optimizations.

In terms of tooling, both Java and Rust have good editors and debuggers, whereas refactoring R code is no better than search and replace and the debugging experience using explicit functions `debug` or `browser` is painful. We also used mature testing libraries in Java and Rust whereas the R version uses `stopifnot` assertions. We found `cargo` to be much more pleasant to use than `maven`.

One issue in Rust is the borrow checker: although the arena allocator made it only a small nuisance, lifetime still shows all around the place. In the GC-ed languages it was completely painless. Garbage collector implementations exist in Rust, but they too add verbosity [6] [1].

Performance. Rust is the fastest, closely followed by Java, and R is the slowest, as expected. The bytecode compilation mostly happens at package installation time. Using a more performant language than R would decrease installation times for large packages, and even for R itself. However, a more performant language would be an additional dependency.

Lessons learnt. Getting the data from and to R (serialization) was the hardest part, and the biggest source of bugs. It was not documented except for the source code. Setting up the infrastructure also took a lot of efforts, as to properly resolve R symbols, we need to boostrap the R environments at the server side. Similarly, constant folding outside of the runtime requires code. Getting the proper representation of opcodes is delicate, from a fully typed one with a complex type hierarchy in Java to a simple ADT and a byte array in Rust.

[4] It is possible to use environments as hash maps.

References

1 Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. Garbage collection makes rust easier to use: a randomized controlled trial of the bronze garbage collector. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1021–1032, New York, NY, USA, July 2022. Association for Computing Machinery. `doi:10.1145/3510003.3510107`.

2 Albert Danial. cloc: v2.04, December 2025. `doi:10.5281/zenodo.5760077`.

3 Pierre Donat-Bouillud, Filip Křikava, Jakob Hain, Adam Plodek, and Jan Vitek. PRL-PRG/r-compile-server. Software, Czech Science Foundation Grant No. 23-07580X, swhId: `swh:1:dir:80fa93336159947c757722b0eea284d792b20055` (visited on 2025-07-16). URL: `https://github.com/PRL-PRG/r-compile-server/tree/main/server/src/main/java/org/prlprg/bc`, `doi:10.4230/artifacts.23610`.

4 Pierre Donat-Bouillud, Filip Křikava, Jakob Hain, Adam Plodek, and Jan Vitek. PRL-PRG/rust-r-bcc. Software, swhId: `swh:1:dir:0dacc419ced335c4daf1d275cb34c26580acdf23` (visited on 2025-07-16). URL: `https://github.com/PRL-PRG/rust-r-bcc`, `doi:10.4230/artifacts.23609`.

5 Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: an IR for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 55–66, 2019. `doi:10.1145/3359619.3359744`.

6 Manish Goregaokar. Manishearth/rust-gc, May 2025. original-date: 2015-05-17T03:31:43Z. URL: `https://github.com/Manishearth/rust-gc`.

7 Andrew L Johnson and Brad C Johnson. Literate programming using noweb. *Linux Journal*, 42:64–69, 1997.

8 Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for R. *ACM SIGPLAN Notices*, 49(7):89–102, 2014. `doi:10.1145/2576195.2576205`.

9 Radford M. Neal. pqr: a pretty quick version of r. `http://www.pqr-project.org/`, 2013. Accessed: 2025-25-03.

10 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. `doi:10.1145/1250734.1250746`.

11 Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. *ACM Sigplan Notices*, 52(2):84–95, 2016. `doi:10.1145/2989225.2989236`.

12 Justin Talbot, Zachary DeVito, and Pat Hanrahan. Riposte: a trace-driven compiler and parallel VM for vector code in r. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 43–52, 2012. `doi:10.1145/2370816.2370825`.

13 Luke Tierney. A byte code compiler for R. *system*, 6:0–010, 2019.