The Journal of Open Source Software

extendr: Frictionless bindings for R and Rust

Mossa Merhi Reimert [1], **Josiah D. Parry** [2], **Matt Denwood** [1], **Maya Katrin Gussmann** [1], **Claus O. Wilke** [3], **Ilia Kosenkov** [4], **Michael Milton** [5], **and Amy Thomason** [6]

1 Section for Animal Welfare and Disease Control, Department of Veterinary and Animal Sciences, University of Copenhagen, Denmark **2** Environmental Systems Research Institute (Esri), Redlands, CA, USA **3** Department of Integrative Biology, The University of Texas at Austin, Austin, TX, USA **4** Independent researcher, Finland **5** Walter and Eliza Hall Institute of Medical Research, Australia **6** Atomic Increment Ltd., United Kingdom

DOI: 10.21105/joss.06394

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @dccsillag
- @alpaylan

Submitted: 06 February 2024
Published: 01 July 2024

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

The programming language Rust continues to gain popularity with developers due to a strong emphasis on safety, performance, and productivity (Perkel, 2020). As a general-purpose, low-level programming language, Rust has a wide variety of potential uses in both commercial and research applications where performance is important. Commercial examples include web development and game development, and in the research domain Rust is increasingly being used in a wide range of contexts including change point detection (Londschien et al., 2023), high-performance GIF encoding (Ooms et al., 2023), and agent-based models of disease spread (Antelmi et al., 2019; Forth et al., 2022; Kshirsagar et al., 2021).

However, typical workflows in research domains, such as disease modelling, often rely on higher-level programming languages due to lower entry barriers. This results in broader adoption within scientific communities, compared to the use of low-level languages like C++ and Rust. The statistical programming language R is one of the most widely used high-level languages in research. R's official interpreter is written in C, and it provides a C API as well as tools for building dynamic libraries in Fortran/C/C++ natively. The 'Extending R' book (Chambers, 2017) also describes interfacing with other languages such as Python and Julia.

The strength of R is its ecosystem of packages, the vast majority of which are available from CRAN. They are primarily written by research scientists, specialists, and professionals. Another important use case of R packages is being a front-end for other languages. Automated tooling that provides scaffolding and boilerplate code is widely used to simplify cross-language integration. For example, embedding C++ code is a good way to resolve performance bottlenecks within R packages, and this can be easily accomplished using cpp11 (Vaughan et al., 2023) or Rcpp (Eddelbuettel & François, 2011). Rust demonstrates similar performance to C++, but it also offers other beneficial features such as declarative memory management, which provides compile-time guarantees for memory safety in the absence of a garbage collector.

We note that other scientific computing communities have already introduced plug-ins for Rust, including Python via PyO3, and Julia via jlrs.

This paper introduces a collection of four Rust crates and an R package that collectively make up the 'extendr' project. The goal of this project is to provide (automatic) binding of Rust to R, using an opinionated and ergonomics-focused suite of tools that facilitate the use of Rust code within R packages. This is achieved by offering emulation of the R data model within Rust, integration of Rust tooling in the R-package build systems, a Rust developer experience in R, and functions for preparing Rust-powered R-packages for submission to CRAN. An overview of the 'extendr' crates and packages as well as comprehensive API documentation

is available at extendr.github.io.

Statement of Need

R provides tools for compiling and embedding Fortran, C, and C++ code, with binding through R's C-API. However, these raw bindings are not easy for users to navigate. This makes frameworks facilitating interfacing other languages to R extremely popular. Rcpp (Eddelbuettel & François, 2011) and cpp11 (Vaughan et al., 2023) for C++, Java via rJava (Urbanek, 2023), Python with `reticulate` (Ushey et al., 2023), and JavaScript on the V8 runtime and the V8 R-package (Ooms, 2023) are among the most used. In contrast, bindings between Rust and R, such as `gifski` (Ooms et al., 2023), are currently mostly written by hand.

We note that other software packages providing bindings between R and Rust exist. The Rust crate / R-package `roxido` / `cargo` (Dahl, 2021) provides a mechanism for embedding Rust code within R packages. The savvy interface represents a distilled byproduct of 'extendr'. However, 'extendr' differs from these implementations in that 'extendr' aims at providing an opinionated API, with a focus on an ergonomic API design inspired by features from Rcpp and cpp11.

Several existing projects already utilize 'extendr'. The DataFrame library Polars has bindings to Python (via `py-polars`) and to R via `polars`, where the latter is built with 'extendr'. The CRAN package `rsgeo` provides bindings to `geo-rust`, allowing R users to take advantage of highly performant geometric primitives and algorithms written and optimized in Rust.

Another example of scientific work enabled by 'extendr' is the `changeforest` package (Londschien et al., 2023).

Design

Overview

The extendr project provides a suite of software packages, where the aim is to provide a mechanism for interfacing Rust to R that is comparable in scope to the R/C++ interfaces provided by Rcpp and cpp11. It consists of the following components:

- `extendr-api`: a Rust crate integrating R's data model in Rust, which underlies the functionality of extendr
- `extendr-macros`: a Rust crate responsible for auto-generating R wrappers for embedding Rust within R code
- `extendr-engine`: a Rust crate that enables launching R sessions from within Rust code, similar to `RInside` (Eddelbuettel et al., 2023)
- `rextendr`: an R package that simplifies the process of embedding Rust code within an R package, including helping the user to adhere to CRAN rules for publishing Rust-powered R packages
- `libR-sys`: a Rust crate providing auto-generated Rust bindings to R's C-API

Using 'extendr' requires both Rust and R to be installed, but no other dependencies are required. API documentation for all the 'extendr' packages are available at extendr.github.io, and the repositories for 'extendr'-packages are freely available from GitHub github.com/extendr, under an MIT license. All hardware/software platforms supported by Rust and R are also supported by extendr.

Technical details

Most R data is vector-based, including scalar values (which are length-1 vectors). These vectors are represented in Rust as slices `&[T]` / `&mut [T]`. R data may be allocated in Rust,

but these are invisible to R's garbage collector, and thus have to be protected. `extendr-api` registers R allocated data in an internal hash-map / dictionary, that stores a reference count for Rust allocated R data.

A C-function is callable in R if it returns an `SEXP` and all of its arguments are `SEXP` - these are opaque pointers to an internal R representation of data. These are callable in R via `.Call`. A Rust function that is exported to R must have all of its arguments and return values convertible to `SEXP`. Annotating it with `#[extendr]` will add a callable C-function in R that converts the custom data types into `SEXP` types.

The `rextendr` package also provides R-level functions: `rust_source`, which allows arbitrary Rust code to be evaluated, returning the last value in the block; and `rust_function`, which compiles, wraps, and returns arbitrary Rust functions as callable R functions. These two functions are very similar in scope to the `evalCpp` and `cppFunction` functions provided by Rcpp, and are very versatile, as they can also be used to include third-party crates.

Creating Rust-powered R packages

The `rextendr::use_extendr()` function can be used to auto-edit an existing user-specified R package (for example created using `usethis::create_package()`) to include all of the details necessary to embed Rust code within the package. This includes Makevars files that adapt the compilation process of the R package to generate the embedded Rust binary using R's internal build system.

This should then be followed by calling `rextendr::document()`, which provides R wrapper functions, within which the Rust functions are invoked via the `.Call` foreign function interface.

For many R package authors, being able to publish their code on CRAN is essential. However, CRAN has strict rules for publishing packages, including that the number of threads that a package uses at build & testing must not exceed 2. Uniquely, Rust has a package manager, which means that R packages have third-party dependencies external to R and CRAN. These must be "vendored" to ensure package stability (see "Using Rust in CRAN packages"). The `rextendr::use_cran_defaults()` and `rextendr::vendor_pkgs()` will ensure that dependencies are built entirely offline and from vendored sources, which ensures that the resulting R package is fully CRAN-compliant.

Getting started

Ensure that both R and Rust are installed. Then in an R terminal, the `rextendr` package can be installed via:

```
install.packages("rextendr")
```

Or, for the latest development version:

```
remotes::install_github("extendr/rextendr") # installs latest dev-version
```

Then, an R-package should be constructed - optionally using the `usethis` R-package (Wickham et al., 2023), which inspires the design principles of `rextendr`:

```
usethis::create_package("exampleRustRpkg")
rextendr::use_extendr()
```

Finally, the function `use_extendr` should be used to set up the necessary boilerplate for compiling Rust code within an R package, and `document` used to refresh the R function wrappers (this augments a call to `devtools::document()`).

```
rextendr::document()
```

Acknowledgements

Project lead Amy Thomason received a grant from the R-consortium (R Consortium, 2023).

Mossa Merhi Reimert received funding from the Danish Food and Veterinary Administration for his PhD project.

Claus O. Wilke acknowledges funding from the University of Texas at Austin (Reeder Centennial Fellowship in Systematic and Evolutionary Biology, Blumberg Centennial Professor in Molecular Evolution).

We would like to acknowledge Jeroen Ooms for his hellorust (Ooms & Authors of the dependency Rust crates, 2023), and continuous maintenance of a hand-written embedding of Rust in an R proof-of-concept project. Their github.com/r-rust contains several examples of hand-crafted bindings to Rust packages for R, such as `gifski` (Ooms et al., 2023).

References

Antelmi, A., Cordasco, G., D'Auria, M., De Vinco, D., Negro, A., & Spagnuolo, C. (2019). On evaluating Rust as a programming language for the future of massive agent-based simulations. *Communications in Computer and Information Science*, *1094*, 15–28. https://doi.org/10.1007/978-981-15-1078-6_2

Chambers, J. M. (2017). *Extending R*. CRC Press. https://doi.org/10.1201/9781315381305

Dahl, D. B. (2021). *Writing R extensions in Rust*. https://arxiv.org/abs/2108.07179

Eddelbuettel, D., Francois, R., & Bachmeier, L. (2023). *RInside: C++ classes to embed R in C++ (and C) applications*. https://doi.org/10.32614/cran.package.rinside

Eddelbuettel, D., & François, R. (2011). Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, *40*(8), 1–18. https://doi.org/10.18637/jss.v040.i08

Forth, J. H., Calvelage, S., Fischer, M., Hellert, J., Sehl-Ewert, J., Roszyk, H., Deutschmann, P., Reichold, A., Lange, M., Thulke, H.-H., Sauter-Louis, C., Höper, D., Mandyhra, S., Sapachova, M., Beer, M., & Blome, S. (2022, September 8). *African swine fever virus – variants on the rise*. https://doi.org/10.1101/2022.09.07.506908

Kshirsagar, J. K., Dewan, A., & Hayatnagarkar, H. G. (2021). EpiRust: Towards a framework for large-scale agent-based epidemiological simulations using Rust language. *SIMS Conference on Simulation and Modelling (SIMS 2020)*, 475–482. https://doi.org/10.3384/ecp20176475

Londschien, M., Bühlmann, P., & Kovács, S. (2023). Random forests for change point detection. *Journal of Machine Learning Research*, *24*(216), 1–45. https://doi.org/10.3929/ethz-b-000585774

Ooms, J. (2023). *V8: Embedded JavaScript and WebAssembly engine for R*. https://doi.org/10.32614/cran.package.v8

Ooms, J., & Authors of the dependency Rust crates. (2023). *hellorust: Minimal examples of using Rust code in R*. https://doi.org/10.32614/cran.package.hellorust

Ooms, J., Kornel Lesiński, & authors of the dependency Rust crates. (2023). *Gifski: Highest quality GIF encoder*. https://doi.org/10.32614/cran.package.gifski

Perkel, J. M. (2020). Why scientists are turning to Rust. *Nature*, *588*(7836, 7836), 185–186. https://doi.org/10.1038/d41586-020-03382-2

R Consortium. (2023, July 6). *R Consortium funded project extendr provides Rust extensions for R*. R Consortium. https://www.r-consortium.org/blog/2023/07/06/r-consortium-funded-project-extendr-provides-rust-extensions-for-r

Urbanek, S. (2023). *rJava: Low-level R to Java interface*. https://doi.org/10.32614/cran.package.rjava

Ushey, K., Allaire, J., & Tang, Y. (2023). *reticulate: Interface to Python*. https://doi.org/10.32614/CRAN.package.reticulate

Vaughan, D., Hester, J., & François, R. (2023). *cpp11: A C++11 interface for R's C interface*. https://doi.org/10.32614/cran.package.cpp11

Wickham, H., Bryan, J., Barrett, M., & Teucher, A. (2023). *usethis: Automate package and project setup*. https://doi.org/10.32614/cran.package.usethis