Writing R Extensions in Rust

David B. Dahl
Department of Statistics
Brigham Young University
dahl@stat.byu.edu

1 Introduction

Novel methods in data science and statistics with supporting software often have a broader impact than methods introduced without software. Supporting software is typically written in a high-level language, with performance-critical parts calling libraries written in C, C++, FORTRAN, etc. We recommend Rust for performance-critical components of a Python or R package, and we specifically address developing Rust-based R packages in this paper. Rust “empowers everyone to build reliable and efficient software” as performant as C/C++. (See The Computer Language Benchmarks Game.) Rust has been rated the “most loved programming language” in the Stack Overflow Annual Developer Survey every year since 2016.

This paper complements *Writing R Extensions*, the official guide for writing R packages, for those who are familiar with Rust and want to write a Rust-based R package. An R package named `cargo` is on CRAN to aid development. It provides complete bindings for all of R’s API, but idiomatic Rust functions are also available and often avoid the need to directly call R’s API. Examples on R packages on CRAN developed using the `cargo` package include `salso`, `caviarpd`, and `fangs`.

The `rextendr` package provides another approach to develop Rust-based R packages. It aims to provide extensive automatic conversion between R types (e.g., vectors, lists, data.frames, etc.) and Rust types, including handling thorny issues such as R’s missing value `NA` and R’s fluidity in the storage mode of vectors. The advantages of the `cargo` package’s approach include its transparency, low overhead, extensibility, and CRAN policy compliance.

We assume the toolchain for building R packages is installed. Use RTools on Windows and follow these instructions on MacOS. Also, install the `cargo` package from CRAN using `install.packages("cargo")` and install the Rust toolchain as show in `file.show(system.file("template/INSTALL", package="cargo"))` or use `cargo::install()`.

2 Methods

2.1 Getting started

Start a new Rust-based R package using, for example, `cargo::new_package("/path/to/package/foo")` to generate the `foo` package. Or, in RStudio, select "File → New Project... → New Directory → Rust-based R Package (using cargo package)". The resulting package is a complete R package with the typical directory structure, plus a few Rust-specific items. The binary package does not depend on Rust.

2.2 Calling a Rust function

Note that there are several uses of `.Call()` among the scripts in the `R` directory. The function in `R/myrnorm.R`, for example, has `.Call(.myrnorm, n, mean, sd)` which executes the Rust function `myrnorm` defined in `src/rust/src/lib.rs`:

```
1  mod registration;
2  use roxido::*;
3
4  #[roxido]
5  fn myrnorm(n: Rval, mean: Rval, sd: Rval) -> Rval {
6    unsafe {
7      use rbindings::*;
8      use std::convert::TryFrom;
9      let (mean, sd) = (Rf_asReal(mean.0), Rf_asReal(sd.0));
10     let length = isize::try_from(Rf_asInteger(n.0)).unwrap();
11     let vec = Rf_protect(Rf_allocVector(REALSXP, length));
12     let slice = Rval(vec).slice_mut_double().unwrap();
13     GetRNGstate();
14     for x in slice { *x = Rf_rnorm(mean, sd); }
15     PutRNGstate();
16     Rf_unprotect(1);
17     Rval(vec)
18   }
19 }
```

All Rust functions with the `#[roxido]` attribute take arguments of type `Rval` and return a value of type `Rval`. Among other things, the `#[roxido]` attribute wraps the body of the function in a call to Rust’s `std::panic::catch_unwind` since unwinding from Rust code into foreign code is undefined behavior and will likely crash R. When a panic is caught, it is turned into an R error, showing the corresponding message in the R console and giving the line number. The package developer is encouraged to study the definition of the `#[roxido]` attribute in `src/rust/roxido_macro/src/lib.rs`.

2.3 Low-level interface to R’s API

The `myrnorm` function above illustrates how to directly use R’s API in Rust. Note that the statement `use rbindings::*` provides direct access to R’s API through Rust bindings. These are automatically generated by the bindgen utility from R header files. The documentation for the Rust bindings can be browsed by running `cargo::api_documentation()` when the current working directory is the package root. Note that most of the functions in the `rbindings` module require an `SEXP` value, i.e., a pointer to R’s internal `SEXPREC` structure. When calling R API functions, the `SEXP` must be extracted from an `Rval` value, e.g., `mean.0` as in line 9. Conversely, when returning from a function marked with `#[roxido]` attribute, wrap the `SEXP` value `x` in `Rval(x)`, as in line 17.

When calling an R API function, care should be taken so that the R function does not throw an error. Otherwise, a long jump occurs over Rust stack frames, preventing Rust from doing its usual freeing of heap allocations and leaking memory. Care must also be taken when calling R API functions that might catch a user interrupt because an interrupt also produces a long jump. One R API function that catches interrupts, for example, is the `Rprintf` function for printing to R's console.

2.4 High-level interface wrapping R's API

To avoid the pitfalls of directly accessing R API functions and to provide a more idiomatic Rust experience, the `cargo` package also provides a high-level interface defined in the `r` module. The high-level interface is not a comprehensive wrapper over R's API, but it covers common use cases and the developer can easily expand it by adding to `src/rust/roxido/src/r.rs` in the package. The high-level interface provides the `check_user_interrupt` function. The `rprintln!` macro is analogous to Rust's standard `println!` macro, but prints to the R console and returns `true` if interrupted. Much of the interface is provided by associated functions for the `Rval` structure. For details, see the documentation using `cargo::api_documentation()`.

The package generated by the `cargo::new_package` function provides examples of the high-level interface. Consider the `convolve2` function from Section 5.10.1 "Calling .Call" of *Writing R Extensions*. A Rust translation based on the `cargo` package is in `src/rust/src/ lib.rs` and shown here:

```
1  #[roxido]
2  fn convolve2(a: Rval, b: Rval) -> Rval {
3      let (a, xa) = a.coerce_double(pc).unwrap();
4      let (b, xb) = b.coerce_double(pc).unwrap();
5      let (ab, xab) =
6          Rval::new_vector_double(a.len() + b.len() - 1, pc);
7      for xabi in xab.iter_mut() { *xabi = 0.0 }
8      for (i, xai) in xa.iter().enumerate() {
9          for (j, xbj) in xb.iter().enumerate() {
10             xab[i + j] += xai * xbj;
11         }
12     }
13     ab
14 }
```

Notice on lines 3 and 4 the calls to `Rval`'s `coerce_double` method. The method returns either a tuple giving a (potentially new) `Rval` and an `f64` slice into it, or an error. A slice into R's memory for vectors of doubles, integers, and logicals can be obtained without a potential memory allocation using `x.slice_*()` or `x.slice_mut_*(x)`, where `x` is an `Rval`.

Notice the argument to the `coerce_double` method on lines 3 and 4 is `pc`. The wrapper code provided by the `#[roxido]` attribute includes `let pc = &mut Pc::new()`. Many of the functions take a shared mutable reference to a `Pc` structure. The purpose of `Pc` is to handle the bookkeeping associated with `Rf_protect` and `Rf_unprotect` calls related to R's garbage collection. When an instance of the `Pc` structure goes out of scope, the Rust compiler automatically inserts a call to its associated `drop` function which executes `Rf_unprotect` using its interval protect counter. Not only does the developer not need to manually track the number of protected items, the developer does not need to worry about when a value should be protected. If the method requires a shared mutable reference to a `Pc`, then protection is needed and automatically handled.

2.5 Embedding Rust code in an R script

Beyond package development, the `cargo` package also supports defining functions by embedding Rust code directly in an R script. This facilities quick experimentation and testing. See the documentation for the `cargo::rust_fn` function.

2.6 Workflow

The `DESCRIPTION` file has `SystemRequirements: Cargo (>= 1.XX)...`, where `1.XX` is a version number, and this should be updated when your code use features from a more recent version of Rust. (One can use `cargo-msrv` to find the minimum supported Rust version.) As CRAN machines may only occasionally update their Rust installation, one should be somewhat conservative in adopting new Rust features.

The `configure` script compiles the Rust code in `src/rust` to a static library using the `run` function from the `tools/cargo_run.R` script. For the sake of CRAN policy compliance, notice that the `run` function is configured to use a temporary directory, only use two cores, and run in offline mode. The static library is folded into the package's shared library through the `src/Makevars` file.

The `cargo::prebuild` function provides tools for package maintenance. When Rust dependencies are updated in the `src/rust/Cargo.toml` file, set the working directory to the package root and run `cargo::prebuild(c("authors","vendor"))`. This updates `src/rust/vendor.tar.xz` and, to help in manually updating the `Authors@R` field of the `DESCRIPTION` file, creates the file `authors-scratch.txt`. When `roxygen2` documentation is updated in a package's R script, run `cargo::prebuild("document")`. When wanting to make a new Rust function accessible from R (or when updating the number of arguments to a Rust function), add the appropriate `.Call` to the package's R code and run `cargo::prebuild("register_calls")`.

3 Discussion

We end with a discussion of a few miscellaneous points to keep in mind when developing. Care should be taken when dealing with R's special values. For example, R's `NA` integer value corresponds to Rust's `i32::MIN`. So, `NA_integer_ * 0L` in R equals `NA_integer_` but equals `0` in Rust. Associated functions, such as `Rval::is_na_integer`, are provided to test against R's special values. See Section 5.10.3 "Missing and special values" of *Writing R Extensions* for a discussion.

Rust supports "fearless concurrency," making it safe and easy for Rust-based R packages to harness the power of multiple CPU cores. R's internals are fundamentally designed for single-threaded access, however, so any callbacks into R should come from the same thread from which R originally called the Rust code.

R users expect reproducible results when using R's `set.seed` function. Options are: (i) produce random numbers using R's API (as in the previous `myrnorm` example) or (ii) seed a Rust random number generator from R's random number generator using the provided `random_bytes` function. For example, to seed `Pcg64Mcg` from the `rand_pcg` Rust crate, use `Pcg64Mcg::from_seed(r::random_bytes::<16>())`.