Rust's Foreign Function Interface

Srinath Kailasa [*]
University College London

November 8, 2022

Foreign Libraries from Rust

Rust is currently unable to call directly into a C++ library. Can use the snappy C library interface to call into from Rust.

Rely on the 'libc' crate, which provides type definitions for C types, among other things such as function headers e.g. malloc. and constants e.g. 'EINVAL'.

Start off with 'extern' block, contains a list of foreign function signatures. The '#[link(...)]' attribute is used to instruct the linker to tlink against the snappy library so the symbols are resolved.

Rust's compiler cannot check the function argument types of foreign functions. So it's up to a user to correctly resolve these.

Must wrap the raw C API with Rust functions to provide memory safety, and use Rust concepts such as vectors.

Foreign libraries often hand off ownership of resources to the calling code. When this occurs, we must user Rust's destructors to provide safety and guarantee the release of these resources. (Drop trait).

Rust from Foreign Libraries

Use the C ABI to expose Rust to other languages, easy as long as they also can interact with libraries via their C ABIs.

[*]srinath.kailasa.18@ucl.ac.uk

The ‘extern ”C” ’ keyword makes the function adhere to the C calling convention. O.e. how program parameters are pushed on the stack by the caller from right to left. The caller is in charge of cleaning up the stack after the call.

In computer science, a calling convention is an implementation-level (low-level) scheme for how subroutines receive parameters from their caller and how they return a result. Differences in various implementations include where parameters, return values, return addresses and scope links are placed (registers, stack or memory etc.), and how the tasks of preparing for a function call and restoring the environment afterwards are divided between the caller and the callee.

- How the stack frame for the call is setup
- How the arguments are passed, whether on the stack or in registers, or in reverse order.
- How the function is told where to jump on return
- How various CPU states like registers are restored in the caller after the function completes.

Also use ’nomangle’ attribute, to turn off Rust’s name mangling to that we have a well defined symbol to link to. We compile the Rust code as a shared library, using the key value ’cdylib’ (can also compile as a static lib if needed).

gcc call_rust.c -o call_rust -lrust_from_c -L./target/debug/ -arch x86_64

The link attribute on extern blocks provides the basic building block for instructing rustc how it will link to native libraries. There are two accepted forms of the link attribute today:

link(name = ”foo”)

(name = ”foo”, kind = ”bar”)

foo is the name of the native library we’re linking to, and in the second case bar is the type of library. Either dynamic, static, (and ’framework’ for macos targets)

FFI, is all about accessing foreign bytes outside of application’s Rust src. For that, Rust provides two primary building blocks. ‘Symbols’ and ‘calling conventions’. Symbols are names assigned to particular addresses in a segment of your binary. that allows you to share memory (be it for data or code) between the external origin and Rust code. Calling conventions that that provide a common understanding of *how* to call functions stored in shared memory.

Generic functions may even have multiple symbols, one for each monomorphization.

Symbols are normally used internally by the tocmpiler to figure out the final address of a function or static variable in your binary. Different function calls can have different symbols, therefore we need to have stable names for symbols when using FFI.

Compilation Redux

Three distinct phases: compilation, code generation, linking. Handled often by different programs.

The first phase is what’s traditionally thought of as a compiler. Checks types, borrow checking, monomorphization etc. Generates some IR that’s passed to the code generation tool. Can compile chunks of src in parallel.

Linker’s primary job is to take binary artifacts (object files) generated by code generator and stich together final binary, inlcuding references to third party libraries. I.e. stitches together all final memory addresses too.

The linker is what enables FII to work. It doesn’t care about how each input object files were constructed, only about resolving shared symbols. The object files could have come from any language. Static linking - copy and paste into binary. Dynamic linking - paste reference.

Maturin

A tool to build and publish crates using pyo3, rust-cpython, and cffi bindings. As well as publish rust binaries as Python packages, either source distributions or built wheels.

Comes with a CLI, to create and build combined Rust + Python projects.

Maturin uses cbindgen to generate a header file to expose Rust lib. Generates a module which exports an ffi and lib object to call from Python.

Can specify Py dependencies via pyproject.toml.

PyO3 Vs rust-cpython

Pyo3 started off as a fork of rust-cpython.

Interface for creating Rust bindings for Python code, as well as tools for creating Py extension modules. i.e. numpy -¿ rust-numpy and rust ¡- Python module.

Rust-CPython is a set of ”Rust bindings for the python interpreter.” Created in 2015, Rust-CPython was the precursor to the PyO3 project. The main difference between the two projects is implementation: in Rust-CPython developers define classes and functions using declarative macros whereas in PyO3, procedural macros are used. An important secondary difference relates to ownership. In PyO3, the framework owns Python values, whereas Rust-CPython code owns its own values. This distinction gives Rust-CPython users greater flexibility at the cost of some overhead.

https://depth-first.com/articles/2022/03/09/python-extensions-in-pure-rust-with-rust-cpython/

Both offer higher level APIs for wrapping Rust types and functions and structs into Python equivalents. For our purposes, we prefer CFFI - simplest to do, just have to manually match types. We’re passing simple data - just pointers to arrays.

CBindgen

Automatically generates Rust FFI bindings to C and C++ libraries

Rust Build Scripts

Some packages need to compile third-party non-Rust code, e.g. C libs. Other packages need to link to C libs, which are either installed, or need to be built from source.

‘build.rs’ will cause Cargo to compile that script and execute it before building the package.

Can do things like:

- Build a bundled C library
- Finding a C library on the host
- Generating a Rust module from a spec

- Performing platform specific configuration

build scripts are understood by Cargo.

related crates:

cc, access to C/C++ compiler (assembly too). pkg-config, detect's system libraries using the pkg-config linux utility (.pc files). cmake, can run cmake util from Rust