COMS W6156 - Topics in Software Engineering

Spring 2019

Project Report

Bruno Martins - bm2787

Introduction

Rust is a new (version 1.0 was released in 2015) systems programming language developed by Mozilla as a fast and safe alternative to the dominant C and C++ languages. In their words: "Rust is a systems programming language that runs blazingly fast, prevents almost all crashes, and eliminates data races". While most of the language features borrow judiciously from other languages, there is one central novel feature that seems to be unique to Rust: the borrow checker. The borrow checker statically evaluates the ownership and the lifetime of all pieces of data in a program to find and prevent entire classes of bugs like data races and use-after-free.

My project expanded on the work I started for my midterm paper in order to explore more of Rust's features. The EPICS framework (at version R7.0.2.1) was used as a source of production-quality C/C++ code to be reimplemented in Rust. This project intends to answer **two research questions**:

1) Is it feasible to port parts of a big C/C++ project to Rust?
2) Is it worth it?

The **novel** aspect of this project is the use of Rust. The **value** that Rust potentially provides in this scenario is protection against several different classes of bugs, like use-after-free and out-of-bounds array access, while also providing binary compatibility with C and high level abstractions to programmers, such as first class functions, algebraic types, easy concurrency, etc.

Method

The main goal was to reimplement one of EPICS' modules. Initially, I had chosen to reimplement the dbStatic module, written in C, into Rust. The dbStatic module is responsible for parsing files in a Domain Specific Language and creating corresponding in-memory objects that will be used during the execution of an EPICS IOC (Input/Output Controller) program. This module is relatively self-contained, has a well-defined purpose and a number of test files that can be used to validate the eventual Rust implementation. However, I realized that this module has too many lines of code (6.2 kloc according to cloc) and would take too long to be reimplemented.

Therefore, I chose different module, iocsh, to be reimplemented. iocsh is a simple shell that runs inside EPICS IOCs. IOCs, as seen in Figure 1, are responsible for communicating with devices (such as temperature sensors, motor controllers, x-ray detectors...) and making their information available on the network via the Channel Access and/or the PV Access protocols. The responsibilities of iocsh are to parse commands given to it, find the parsed commands in a global registry of available commands, and execute them inside an IOC. This also involves maintaining the global registry of commands and exposing functions to allow different parts of EPICS to register new commands with it. It has only 1.5 kloc and sizable chunks of them can be replaced with Rust's standard library functions.

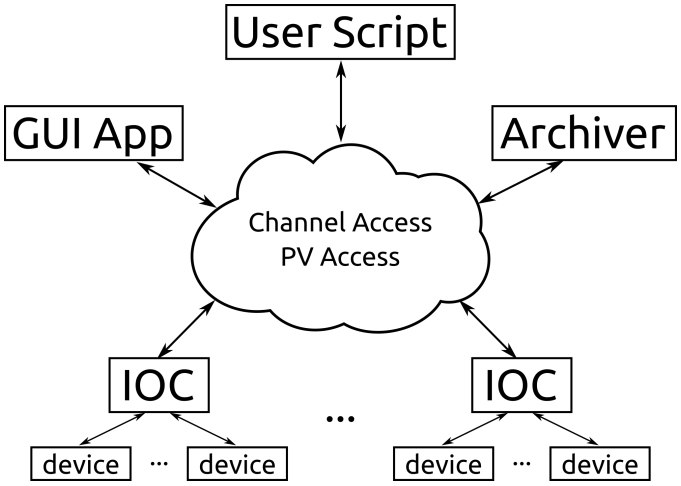

Figure 1: Simplified EPICS architecture

C/C++ to Rust transpilation attempts

In order to try to speed up the development, before I started writing any code I attempted (unsuccessfully) to use a couple of existing C/C++ to Rust transpilers: CRUST and C2Rust.

- **CRUST** was simpler to install and compile, but the program execution "crashed" (panicked, in Rust's parlance) with an "Index out of bounds" error when I tried to use it.
- **C2Rust** is a much more complex and interesting project: it leverages LLVM (and therefore, clang) to do analysis and parsing of the original code. It also uses a tool, called BEAR (Build EAR), that has to be used during a normal compilation of the original EPICS code in order to generate some metadata about the build process itself. Then, C2Rust can use the output of the BEAR program to aid in the transpilation. C2Rust itself took 9 minutes to compile. However, it *also* panicked when I used it, so it is *also* not ready for prime-time. In this case, it might have been due to EPICS relying on a lot of GCC-dependent code, which might not have made LLVM happy.

In the end, I believe that C/C++ and Rust are too different and too complex to have a functioning, general transpiler between them.

Project Development

I spent a considerable amount of time trying different approaches on how to reimplement the shell in a way that preserves functionality and is fully interoperable with C. This project was much more difficult than I had anticipated. For example, the registry that iocsh maintains is implemented, in C, as a program-global hash map *and* a linked list of structures that describe the available commands, which is not protected by a lock (presumably because the registry is only populated during IOC startup, which is single threaded). Rust *does not* like global data like that, *at all*, so I had to fight the compiler for a while until I came up with a working solution. The next difficulty that I faced was the integration between Rust and C code, in the sense of finding exactly how to link (in the linker sense) both Rust-generated and gcc-generated object files together. Since Rust is still fairly new there's a lot of conflicting advice, most being obsolete, on how to approach this issue. However, there's a Rust users forum that seems to be very active. I used it to get some help with the problems I was having.

It is worth noting that this project allowed me to find a bug in EPICS itself (that leads to a segmentation fault, no less), which I filed on EPICS' Issue Tracker (this bug would have been prevented by Rust, and would have been counted in my midterm paper).

Build System Integration

The EPICS framework has a custom build system, built on top of Makefiles, that is capable of compiling EPICS for several different targets: from the usual Linux/Windows/Mac targets to more niche vxWorks and RTEMS embedded operating systems. Rust has a standalone, Rust-specific build system, named cargo, which manages not only the compilation itself but also project dependencies.

The code for this project lives in the `modules/libcom/src/iocsh-rust/` folder. In order to have EPICS be able to compile iocsh-rust I modified `modules/libcom/src/Makefile` to call a simple `Makefile` inside iocsh-rust's folder, which in turn calls cargo. Cargo, then, compiles the Rust code into a C ABI compatible shared object library file, which is *then* linked to the rest of the EPICS framework. It is important to note that, as configured for this project, cargo only compiles for the host architecture (Linux on x86_64) and as a debug (as opposed to a release) target.

With this setup, it is possible to compile both regular EPICS code and the iocsh-rust module at the same time by simply issuing `make` at the root of the project.

Global Data Structures

As mentioned before, iocsh exports functions that allow any part of EPICS to register new commands with the shell. These functions can be called at any time, and the registered commands are kept in a couple of data structures with static storage inside iocsh: a hash map (with a custom C implementation), in order to allow O(1) command lookups, and a linked-list (also with a custom C implementation), in which command description structures are kept in alphabetical order with

respect to the command names, presumably to make it easy to display the list of available commands with the `help` command.

In iocsh-rust I decided to use a single data structure, `HashMap` (available in Rust's standard library). In order to display available commands in alphabetical order, the `help` iocsh command simply sorts the names of the commands in the `HashMap` before printing them to the screen. Rust's compiler does not like variables that are static, global and mutable, because they are not thread-safe. Therefore, in iocsh-rust, I had to put the global `HashMap` behind a `RwLock` (Read-Write Lock), so it can be concurrently read by many threads and be written to in a mutually exclusive way. Rust knows that synchronization primitives, such as the `RwLock`, can be used by different threads safely.

Using C functions in Rust

EPICS attempts to provide many programming constructs to its users as a way to fill the gaps in C's sparse standard library and as a way of allowing EPICS programs to be written in a platform-agnostic way. Examples of such constructs are implementations for a general-purpose hash table and for a linked list, as mentioned before. Modern languages like Rust have such constructs available in their standard libraries. However, one interesting facility provided by EPICS is a macro expansion library, called macLib. macLib allows the users of the IOC shell to be able to parameterize commands by making use of macro expansions. For example:

```
epics> epicsEnvSet("HELLO", "Hello, world!")
epics> echo $(HELLO)
Hello, world!
```

While it would be perfectly possible to reimplement macLib in Rust, it is much better to be able to just use such already available functionality. Hence, this was a great opportunity for taking advantage of Rust's binary compatibility with C. To that end, I declared all available functions in macLib in a way that Rust can understand and use them. I also created thin wrappers around them to act as an interface between safe (Rust) and unsafe (C) code, while also performing data type conversions between the languages. One such noteworthy conversion is between C and Rust strings: a C string is essentially a pointer to a region of memory that has a NULL-byte as its terminating character; a Rust string, on the other hand, has length information encoded in them and no terminating NULL-byte. Also, Rust strings are UTF-8 encoded.

Even though the macLib wrapper (`mac.rs`) that I wrote was very thin, it still clocked at 146 lines of code (as counted by `cloc`).

Command Parser

The main responsibility of the IOC shell is to receive commands from a user or a script and execute them. The syntax for the language that the shell accepts is not formally defined. Instead, command line inputs are parsed in an ad-hoc way. It took a fair amount of time for me to

understand iocsh's grammar. The IOC shell makes use of the widely-used `libreadline` library to provide command-line editing and history capabilities.

In my Rust reimplementation I opted to perform the parsing of commands by making use of regular expressions through the `regex` crate (a crate, in Rust's parlance, is akin to a library in C or Python). My reimplementation also allows command-line editing and history, via the `rustyline` crate. The use of regular expressions greatly simplified the code for parsing inputs to the shell, at a loss of more precise error messages. For example, Table 1 shows the difference in error messages between iocsh and iocsh-rust.

```
epics> echo "Hello
Unbalanced quote.
epics>
```

```
epics> echo "Hello
malformed line
epics>
```

Table 1: Difference in error messages between C (left) and Rust (right) implementations

Exporting Rust functions to C

Since the main objective of this project is to reimplement part of a big C/C++ project in Rust, the resulting Rust module *must* be able to communicate with C/C++ modules. Communications in one direction (Rust accessing C functions) were demonstrated by the use of the macro expansion library, macLib. Communications in the other direction (C accessing Rust functions) are made possible by marking the structures and functions in the Rust module API as C-compatible. This tells the compiler to generate binaries that can be used by C.

This was the bulk of the project and its most challenging part. Passing objects back and forth between C and Rust proved to be difficult due to Rust's great strictness about object lifetimes and access rules, contrasted with C's complete lenience. In many instances it was laborious to determine the ownership of certain resources(and, by extension, which language is responsible for freeing them) coming from C. For example, when registering a command with iocsh, EPICS code allocates static structures and pass pointers to them to iocsh. iocsh, however, is expected to allocate a new structure on the heap that has some more metadata about the command being registered, along with pointers to the passed-in static structures. However, nowhere it is specified that the passed in structures *have* to be static, it is just convention. This kind of implicit lifetime information had to be made explicit to Rust's compiler, which involved a lot of boilerplate and data type conversion code.

I succeeded, however, as shown in the demo given to the class: different parts of EPICS are capable of registering commands with iocsh-rust that can be later executed by iocsh-rust's users.

Functionality that was not implemented

Given the time constraint for finishing this project and obtaining results, I chose not to implement certain iocsh features:

- I/O redirection (reading commands from a file / writing results to a file)
- Access from C code to the structures stored in Rust's command registry (managing type conversion alone would have been very burdensome)

Conclusion

After spending so much time with Rust, I think it is a great and powerful language, with a reasonable standard library and growing ecosystem of crates. I especially enjoyed the strong strictness of Rust's compiler since it gave me confidence that once the code compiled, it would work as expected. Regarding the research questions I set out to answer, here are my conclusions:

1) **Is it feasible to port parts of a big C/C++ project to Rust?**

Yes, as was demonstrated. However, it can be extremely difficult to bridge the gap between the two languages when it comes to making the Rust compiler happy with lifetime and borrowing rules. A lot of effort has to be expended in writing code to convert between Rust and C representations of data. Incidentally, I came across a blog post of a person that did something similar to this project (but in a much larger scale) and came to the same conclusion:

"*Currently there is 11 THOUSAND lines of Rust in wlroots-rs. All of this code is just wrapper code, it doesn't do anything but memory management. This isn't just repeated code either, I defined a very complicated and ugly macro to try to make it easier. This wrapper code doesn't cover even half of the API surface of wlroots. It's exhausting writing wlroots-rs code, memory management is constantly on my mind because that's the whole purpose of the library. It's a very boring problem and it's always at odds with usability*"

2) **Is it worth it?**

In my opinion, no. One can *either* go to the trouble of taking care of every little safety aspect of the interface between C and Rust at great cost in terms of labor *or* use the wrapped C parts as-is, marking its use as unsafe, which would defeat the point of using Rust in the first place.

I learned a few things throughout this project. Mainly, I learned a new programming language, Rust, and I look forward to new opportunities to use it. I also learned a lot more from EPICS internals. Also, perhaps more importantly, I learned a great deal about interfacing Rust and C code in a project, and I feel that this knowledge will be important should I choose to use Rust in the future.

Artifacts

All code for this project, along with instructions on how to compile it and run it, are on GitHub:

https://github.com/brunoseivam/epics-base-rust

This report, along with other milestone reports and the demo slides, are available in the `comsw6156-docs` subfolder.