It's Better to Rust Than Wear Out

GRAEME JENKINSON

Graeme Jenkinson is Senior Research Associate in the University of Cambridge's Computer Laboratory, leading development of distributed tracing for the Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS) project. Prior to working on CADETS, he had 13 years' experience working in the defense and automotive industries.

This article first appeared in the *Free BSD Journal,* Nov/Dec 2016.

When a colleague of mine first enthused to me about Rust, I was skeptical. Back in the day, I'd cut my programming teeth developing software for safety-critical systems, and I'd learned the hard way that programming languages are frequently less sane than they first appear. Take C. Despite a considerable standardization effort, the C specification remains riddled with unspecified, undefined, and implementation-defined behaviors [2]. And even in 2016, researchers continue to explore the differences between the C ISO standard and the de facto usage [4].

While not all software engineers need be concerned with the seemingly esoteric issues of what happens when a bit field is declared with a type other than int, signed int, or unsigned int (it's undefined [2]), I'd worked too long with safety-critical and security systems to switch off this retentive part of my brain. And so, somewhat dismissively, I mentally parked Rust along with Go, Haskell, and all the other technologies that sound cool, but which I could never foresee actually using. Then early this year I had the opportunity to revisit Rust, and I found I'd been a bit hasty.

I had been developing a prototype for a distributed tracing framework built on top of DTrace. The prototype, written in C, acted as a DTrace consumer (interfacing with `libdtrace`) and sent DTrace records upstream for further processing (aggregation, reordering, and so on) using Apache Kafka. For a prototype this worked fine, but as the work progressed, I needed to rapidly explore the design space.

This task favored adopting higher-level language, but which one to choose? Like all good engineers, I started to list out my requirements. I needed a language that emphasized programmer productivity. It needed to easily and efficiently interface with libraries written in C (such as `libdtrace`). I also needed easy deployment, therefore languages requiring a heavy runtime (and Java specifically) were complete nonstarters. Good support for concurrency and, ideally, prevention of data races would be nice. And, finally, with my security hat on, I didn't want to embarrass myself by introducing a bucket-load of exploitable vulnerabilities. I thought back to that earlier conversation with my colleague; aren't these requirements exactly what Rust is designed for? And so I decided to give Rust a whirl, and I'm glad that I did, because I really liked what I found.

So What's Rust All About?

Rust's vision is simple—to provide a safe alternative to C++ that makes system programmers more productive, mission-critical software less prone to bugs, and parallel algorithms more tractable. Rust's main benefits are [5]:

- Zero-cost abstractions
- Guaranteed memory safety (without garbage collection)
- Threads without data races
- Type inference
- Minimal runtime
- Efficient C bindings

The Rust language has a number of comprehensive tutorials, notably the "Rust Book" [5]. Therefore, rather than retreading that ground, I will instead highlight the features of Rust that I find particularly compelling. Along the way, I'll discuss the features of Rust that are most difficult to master. And, finally, I'll show how to get started programming in Rust on FreeBSD.

Fighting the Borrow Checker

Before diving in headfirst and firing up your favorite text editor (`vim`, obviously), it is important to understand Rust's most significant cost, its steep learning curve. On that learning curve, nothing is more frustrating than repeatedly invoking the wrath of the "borrow checker" (the notional enforcer of Rust's ownership system). Ownership is one of Rust's most compelling features, and it provides the foundations on which Rust's guarantees of memory safety are built. In Rust, a variable binding (the binding of a value to a name) has ownership of the value it is bound to. Ownership is mutually exclusive; that is, a resource must have a single owner. It is the borrow checker's job to enforce this invariant, which it does by failing early (at compile time) and loudly.

In the following example, taken from the "Rust Book" (*The Rust Programming Language*, 2016), `v` is bound to the vector `vec![1, 2, 3]`, a Rust macro creating a contiguous, growable array containing the values 1, 2, and 3. The function `foo()` is the "owning scope" for variable binding `v`. When `v` comes into scope, a new vector is allocated on the stack and its elements on the heap; when the scope ends, `v`'s memory (both the components on the stack and on the heap) is automatically freed. Yay, memory safety without garbage collection.

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Ownership can be transferred through an assignment `let x = y` (move semantics). But remember, ownership is mutually exclusive, so in the example below, when the variable `v` is referenced (in the `println!` macro) after the transfer of ownership to `v2`, the borrow checker cries foul: `error: use of moved value: `v`.`

```
let v = vec![1, 2, 3];
let v2 = v;

println!("v[0] is: {}", v[0]);
```

In the next example, calling the function `bar()` passing the vector `v` as an argument transfers the ownership of `v`. When the owning scope, the function `bar`, ends, `v`'s memory is automatically freed as before. Ownership of `v` can be returned to the caller by simply returning `v` from `bar`. This approach would get tedious pretty quickly, and so Rust allows borrowing of a reference (that is, "borrowing" the ownership of the variable binding). A borrowed binding does not deallocate the resource when the binding goes out of scope. This means that after the call to `bar()`, we can use our original bindings once again.

```
fn bar(v: &Vec<i32>) {
    // do something useful v here
}

let v = vec![1, 2, 3];

bar(&v);

println!("v[0] is: {}", v[0]);
```

Immutability by Default

By default, Rust variable bindings are immutable. Having spent many an hour typing `const`, `*const`, and `final` in C and Java, respectively, this feature alone fills me with joy; and what is more, unlike `const`, it actually provides immutability. Variable bindings can be specified as mutable using the `mut` keyword: `let mut x = 10`. Also note the sensible use of type inference. Like variable bindings, references are immutable by default and can be made mutable by the addition of the `mut` keyword (`&mut T`). Shared mutable state causes data races. Rust prevents shared mutable state by enforcing that there is either:

- One or more references (&T) to a resource or
- Exactly one mutable reference (&mut T)

Choosing Your Guarantees

Rust's philosophy is to provide the programmer with control over guarantees and costs. Rust's rule that there can be one or more immutable references or exactly one mutable reference is enforced at compile time. However, in keeping with the overall philosophy, various different tradeoffs between runtime and compile time enforcement are supported.

A reference counted pointer (`Rc<T>`) allows multiple "owning" pointers to the same (immutable) data; the data is dropped and memory freed only when all the referenced counter pointers are out of scope. This is useful when read-only data is shared and it is non-deterministic to when all consumers have finished accessing the data. A reference counted pointer gives a different guarantee (that memory is freed when all owned pointers go out of scope) than the compile time enforced guarantees of the ownership system. However, this comes with additional costs

(memory and computation to maintain the reference count). Similarly, mutable state can be shared (using a `Cell<T>` type); this again brings different tradeoffs for guarantees and costs.

Lifetimes

There is one final and rather subtle issue with ownership. Variable bindings exist within their owned scope, and borrowed references to these bindings also exist within their own separate scope. When variable bindings go out of scope, the ownership is relinquished and the memory is automatically freed. So what would happen if a variable binding went out of scope while a borrowed reference was still in use? In summary, really bad things invalidate Rust's guarantees of memory safety. Therefore, this can't be allowed to happen. Lifetimes are Rust's mechanism to prevent borrowed references from outliving the original resource.

In Rust, every reference has an associated lifetime. However, lifetimes can often be elided. The example below shows equivalent syntax with the lifetime (`'a`) of the reference `s` elided and made explicit:

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded
```

Global variables are likely to be the novice Rust programmer's first interaction with lifetimes. Global variables are specified with Rust's special static lifetime as follows: `static N: i32 = 5;`. A static lifetime specifies that the variable binding has the lifetime of the entire program (note that string literals possess the type `&'static str`, and therefore live for the entire life of the program). If I were to hazard a guess at where lifetimes next rear their heads, it would be storing a reference in a struct. In Rust, a `struct` is used to create complex (composite) datatypes. When Rust `structs` contain references (that is, they borrow ownership), it is important to ensure that any references to the `struct` do not outlive any references that the `struct` possesses. Therefore, a Rust `struct`'s lifetime must be equal to or shorter than that of any references it contains.

Efficient Inheritance

In contrast to C++ and Java's heavyweight approach to inheritance, Rust takes a muted approach; in fact, the word *inheritance* is studiously avoided. With traditional inheritance gone AWOL, classes are no longer needed. Having been freed from the confines of classes, methods can be defined anywhere, and types can have an arbitrary collection of methods. As in Go, inheritance in Rust has been boiled down to simply sharing a collection of method signatures. This approach is sometimes referred to as objects without classes. Rust Traits group together a collection of methods signatures—a Rust type can implement an arbitrary set of Traits. Thus, Traits are similar to `mixins`.

Fighting the Borrow Checker Redux

What makes Rust's ownership system so tricky to master? Ownership is not a complexity introduced by the Rust language; it is an intrinsic complexity of programming regardless of the language being used. Languages that fail to address ownership fail at runtime with data races and so on. In contrast, Rust makes issues of ownership explicit, allowing the language to fail early and loudly at compile time. Rust's borrow checker is like that friend you couldn't quite get on with on first meeting. Over time, and once they've helped you out multiple times, you realize that they've actually got some pretty great qualities and you're glad to have made their acquaintance.

Foreign Function Interface (FFI)

Another of Rust's features that particularly appealed to me is its support for efficient C bindings: calling C code from Rust incurs no additional overhead. Efficient C bindings support incremental rewriting of software, allowing programmers to leverage the large quantities of C code that are not going away anytime soon. External functions fall beyond the protections of Rust and thus are always assumed to be unsafe. It is important to note that there are many behaviors, such as deadlocks and integer overflows, that are undesirable but not explicitly unsafe in the Rust sense.

In Rust, unsafe actions must be placed inside an unsafe block. Inside the unsafe block, Rust's wilder crazier cousin "Unsafe Rust" rules. "Unsafe Rust" is allowed to break limited sets of Rust's normal rules, the most important being that it is allowed to call external functions.

In practice, calling C functions from Rust isn't always quite so straightforward as tutorials make out. Consider calling the function `dtrace_open()` from libdtrace. The C prototype for `dtrace_open()` is shown below:

```
dtrace_hdl_t *
dtrace_open(int version, int flags, int *errp)
{
        ….
}
```

To call `dtrace_open()` from Rust, we first specify the `dtrace_open()`'s signature in an extern block (`extern "C"` indicates the call uses the platform's C ABI). We can then call that function directly from an unsafe block.

```
extern crate libc;

…
```

```
extern "C" {
    fn dtrace_open(arg1: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_int) -> *mut dtrace_hdl_t;
}

fn main() {
    let dtrace_version = 3;
    let flags = 0;
    Let mut err = libc::c_int = 0;
    let handle = unsafe {
        dtrace_open(dtrace_version , flags, &mut err)
    };
}
```

But there is one significant problem: where is the type `dtrace_hdl_t` defined? While `dtrace_hdl_t` can be specified by hand, it contains many, many fields, which in turn use yet more new types that must be defined. Specifying all this by hand would be extremely tedious and error prone. Fortunately, there is a solution. C bindings can be generated automatically using Rust's bindgen crate, `cargo install bindgen`. Unfortunately, `bindgen` is not a very mature tool. And, as a result, manually tweaking its outputs is often required (usually adding or removing mutability). With SWIG (Simplified Wrapper and Interface Generator) support for Rust not looking imminent, better native tooling for generating Rust bindings is desperately needed.

Package Management

The final, and in many ways most important, feature that attracted me to Rust was its support for modern application package management. Rust provides a flexible system of crates and modules for organizing and partitioning software and managing visibility. Rust crates are equivalent to a library or package in other languages, and Rust modules partition the code within the crate.

A Rust program typically consists of a single executable crate, which optionally has dependencies on one or more library crates. Reusable, community-developed library crates are hosted at crates.io, the central package repository for cargo, Rust's package management tool (crates.io is broadly equivalent to Python's PyPI). Rust's `cargo` tool fetches project build dependencies from crates.io and manages building of the software. Yeah, I know, does the world really need yet another mechanism for packaging software, resolving dependencies, and building software? Well perhaps not, but `cargo` actually works really well, though for those with experience with Maven, the bar hasn't been set that high.

Getting Started on FreeBSD

Rust's platform support is divided into three tiers, each providing a different set of guarantees. FreeBSD for x86_64 is currently a Tier 2 platform. That is, it is guaranteed to build but not to actually work. Despite the lack of a guarantee, in practice, things generally seem to work pretty well. Tier 2 platforms provide official releases of the Rust compiler `rustc`, standard library `std` (`pkg install rust`), and package manager `cargo` (`pkg install cargo`). FreeBSD's binary Rust package is currently (at the time of writing) at v1.12 with v1.13 being the latest stable release. Once installed, Rust can be updated to the latest version by executing the rustup script:

```
curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

32-bit FreeBSD sits in Rust's lowly third tier where, without guarantees about either building or working, things are pretty unstable. For example, Rust 1.13 recently shipped in spite of a serious code generation bug on ARM platforms using hardware floating point. Here be dragons, so beware!

Where Are We Now?

Rust started life in 2009 as a personal project of Mozilla employee Graydon Hoare. In subsequent years, Rust has transitioned to a Mozilla-sponsored community project with over 1,200 contributors. Since the 1.0 release, delivered in June 2015, Rust has been used in a number of real-world deployments. June 2016 saw another major milestone on the road to maturity, with Mozilla shipping Rust code for the Servo rendering engine in Firefox 48.

So people are using Rust, but does it really deliver on its vision of providing a safe alternative to C++? I think the answer is pretty much yes, though the differences aren't all that huge. For example, in C++, a `unique_ptr` owns and manages an object and disposes of that object when the `unique_ptr` goes out of scope. Furthermore, ownership can be transferred using `std::move;`, and as a bonus, there is type inference using the `auto` keyword. But in spite of these similarities, smart pointers don't give everything that Rust's ownership system does. In the example below [3], accessing `orig` after the move results in a segmentation fault at runtime—a morally equivalent example in Rust would fail to compile. Failing early is a good thing. That a careful and skilled C++ programmer wouldn't make such mistakes is somewhat of a circular argument, because if such mistakes weren't widespread, languages attempting to prevent them wouldn't exist in the first place. C++ also lacks a module system and has a number of pretty ropey features like header files and textual inclusion. These are all wins for Rust.

```
#include <iostream>
#include <memory>

using namespace std;

int main ()
{
   unique_ptr<int> orig(new int(5));

   cout << *orig << endl;
   auto stolen = move(orig);
   cout << *orig << endl;
}
```

How does Rust compare with C++ on performance? Control studies comparing the performance of idiomatic C++ and Rust are hard to find. A comparison between Firefox's Servo and Gecko rendering engines (written in Rust and C++, respectively) reported that the Rust Servo engine was on the order of twice as fast [1]. While these figures should be taken with a pinch of salt, the consensus opinion is that Rust is at least comparable in terms of performance to C++. One of the reasons for this is that Rust features, like genuine immutability, allow optimizations that can't be made in C++. And Rust's semantics bring significant potential for further optimizations.

Despite the advances made in deploying Rust in production environments, problems remain. The Rust ABI is unstable, and as with the Glasgow Haskell compiler, a stable ABI may never happen, almost certainly not anytime soon. This problem most impacts Rust native, shared libraries because without a stable ABI, they are incompatible across major version changes. But ABI instability isn't a showstopper. So is there a technical barrier to upstreaming Rust code to FreeBSD, for instance? In my opinion, I don't think so, but I'd be interested to hear others' opinions on both the technical and political challenges of doing so.

I like Rust. It's fun. And isn't that what really makes us come into work in the morning?

References

[1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the Servo Web Browser Engine Using Rust," in *Proceedings of the 38th International Conference on Software Engineering Companion* (May 2016), pp. 81–89.

[2] L. Hatton, *Safer C*, 1st ed. (McGraw-Hill, 1995).

[3] S. Klabnik, Unique Pointer Problems, Steve Klabnik's home page: http://www.steveklabnik.com/uniq_ptr_problem/.

[4] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell, "Into the Depths of C: Elaborating the De Facto Standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2016), pp. 1–15.

[5] *The Rust Programming Language*, "Getting Started": https://doc.rust-lang.org/book/getting-started.html.